Implementation of an SQL to IQL Query Translation Component for the AutoMed Toolkit

Jamie T. Walters

MSc Advanced Information Systems Project Report

School of Computer Science and Information Systems

Birkbeck, University of London

2007

I. TABLE OF CONTENTS

II. TABLE OF FIGURES

III. LIST OF TABLES

IV. ABSTRACT

IQL as AutoMed's query language presents itself as a potential barrier to users. As a functional language it is more expressive than SQL but is also more complicated to learn and understand. Its lower level implementation however, allows for it to be readily translated to and from the higher level query languages. This project, through background research, sought to establish the most suitable approach towards implementing a solution for translating input queries from SQL to IQL. As the most suitable approach, JavaCC, a parser generator was used to specify the grammar and translation rules. The complexity of these rules varied with the types of queries being translated, however the underlying support for relational algebra in both IQL and SQL created a common base for the translation process. Although the resulting translator supports only subset of SQL queries, its modularity allows for it to be readily extended to support any subset ANSI compliant SQL.

V. CORRECTION PAGE

1 INTRODUCTION

1.1 Description

The AutoMed Heterogeneous Data Integration System (AutoMed) developed by Birkbeck and Imperial College, is an advanced heterogeneous data integration system implemented in Java using a functional query language, IQL. IQL is a potential obstacle for users more familiar with the commonly used Structured Query Language (SQL). This obstacle could be removed if users were able utilize the AutoMed toolkit while using SQL. This project will develop a translator of SQL into the IQL language used for executing queries on the AutoMed system (1).

1.2 Objectives

The main objective of this project is to develop an SQL-to-IQL query translator component for the AutoMed toolkit. This will benefit AutoMed in two ways. Firstly, AutoMed will be more easily used and learned as users will not need to learn a new query language. Secondly, it will make it easier for AutoMed to interoperate with other existing SQL-based systems.

1.3 Methodology

1. **Obtain an understanding of the AutoMed system**

   This will be achieved through literature review of the technical reports on AutoMed. There will be a review of journals and conference reports from AutoMed developers. Further understanding will be gained through examining the source libraries and practical work.

2. **Examine the structure of IQL queries**

   IQL queries will be examined through review of AutoMed technical reports along with practical exercises using the sample data distributed with AutoMed. *A Tutorial on the IQL Query Language* (2)will be taken as the current technical report on the IQL language.

3. **Examine the structure of SQL statements.**

SQL statements will be examined through a review of journals and books. Being the de facto standard for queries, a thorough knowledge on the structure of SQL can be gleaned from its grammar definition and BNF specification.

4. **Investigate previous work on translating from other query languages into functional query languages.**

   Literature review of any previous work done on query translations as well as any test suites, source codes, procedures and problems encountered. Any previous work in the area would be critically reviewed to refine the approach needed for this project.

5. **Explore ways of translating between the structures of SQL and IQL**

   The previous step would provide a starting point for exploring the ways of implementing the translation. Previous works on translating queries as well as the current environment in which AutoMed is used are factors which need to be examined in order to decide on a suitable means of translation.

6. **Understand the use of parser generators and identify the most notable parser generator tools currently available.**

   Parser generators are relatively underutilized development tools. Literature review and basic practical use will be needed in order to determine the most suitable tools available.

7. **Specify a parser generator specification for producing a translator that will translate SQL to IQL Queries.**

   This step is entirely practical. The grammar for the translator will be specified incrementally and the parser developed in stages. This will allow the developer to fully appreciate the issues involved in creating a parser. This resulting modular parser that can then be readily extended as support for different SQL features are added.

8. **Test and improve the translator.**

   Test the accuracy of the translator in identifying components of a SQL statement and producing an accurate IQL equivalent. The translator is continually checked and improved to achieve maximum accuracy.

9. **To develop a suite of SQL test queries to (a) test the correctness of the translator component (b) compare user SQL queries submitted to an AutoMed global schema to the SQL queries submitted to local relational DBMSs**

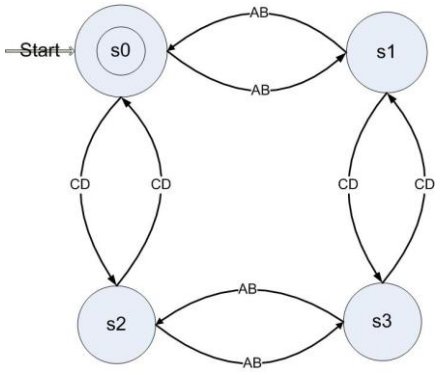
A test suite of different SQL statement types, conforming to ANSI SQL 1992 standard will be developed to test the correctness of the input queries after being translated to the source databases. The test suite should be comprehensive but not overly exhaustive as there is a time constraint on the development of the translator. The results, with actual data, can be further used to improve the accuracy of the translator.

2 BACKGROUND RESEARCH

2.1 Automata

Automata Theory is the study of the abstract mathematical models of machines and the problems they are capable of solving. Automata are used to study and test the limitations of computation and focus on the *decidability* and the *intractability* of computers. *Decidability* tests whether or not a problem can be solved while *intractability* determines the efficiency of the computer with respect to the increase in size or complexity of the problem.

An automaton is a self operating electronic or mechanical device. An automaton consists of a finite set of internal states which accept and respond to an external stimulus to produce an output based on the transition rules applied to the internal states.



2.1 - Simple Automaton

Figure 2.1 is an example of a simple automaton. It accepts an input at *s0* or the start state and switches to a new state within the automaton based on the evaluation of the stimulus. The output of each state determines the next state of the automaton. Figure 2.1, can be classified as a finite automaton as there are a finite number of possible states (4) of the automaton. Finite automata define a class of automata for which there exists a finite number of states, transitions and actions.

The main concepts of the automata theory are alphabet, strings and languages. The alphabet may be defined as some finite set of symbols; strings as a finite sequence of symbols derived from this alphabet and the language as a set of strings such that all the symbols of the strings are derived from this alphabet. In Figure 2.1, the alphabet is {A, B, C, D} while the strings, as shown could be {AB}, {C D} or any combination of members of the alphabet.

Alan Turing attempted to define the boundaries of computing machines and created Turing machines which exist as the most general form of automata. These machines could compute the same types of algorithms as modern computers and as such, his tests and theories still apply to modern computation. Turing machines can simulate the functionality of modern computers which implement a category of automata called finite automata. These machines implement algorithms defined in regular languages and grammars and are employed in the design and implementation of modern computing languages and software solutions.

2.2 Grammars

Precisely defining natural languages has long been regarded as the primary work of linguist Noam Chomsky. Chomsky sought to use mathematical representation to create a structured, flexible and defined model for natural languages. One of his approaches, context-free grammars, was later adopted for modelling computer languages where a grammar may be defined as:

… A quadruple (∑; V; S; P), where:

1. ∑ is a finite nonempty set called the terminal alphabet. The elements of ∑ are called the terminals.

2. V is a finite nonempty set disjoint from ∑. The elements of V are called the non terminals or variables.

3. S ∈ V is a distinguished non terminal called the start symbol.

4. P is a finite set of productions (or rules) of the form

$$\alpha \rightarrow \beta$$

where α ∈ $(\sum \cup V)^*V(\sum \cup V)^*$ is a string of terminals and non terminals at least one non terminal and β is a string of terminals and non terminals. (3)

There are four distinct types of grammars: type 3 or regular grammars, type 2 or context free grammars, type 1 or context sensitive grammars and type 0 or unrestricted .This research is focused on parsers created using context free grammars.

A Context-free grammar (CFG), more generally called a grammar is such that every production is represented by the Backus-Naur form:

$$V \rightarrow t$$

Where *V* is a non terminal and *t* is a terminal symbol from the language's alphabet. Context free grammars allow for flexible syntax structures and the definition strings in natural language. This is then converted into the strict mathematical structures envisioned by Backus (see section 2.3).

Productions in CFG are generally in one of the two approaches; recursive inference or derivation. In recursive inference, strings of the language body variables are concatenated with any terminals in the body. From this it is inferred that the resulting string is in the language of the variable in the head. Derivation involves expanding the start symbol and all subsequent strings by substituting variables with a production until the string is comprised entirely of terminals.

Derivation grammars exist as either Leftmost (LL) grammars or rightmost LR grammars. In LL grammars, the production rules are applied on the left most variable in the body of a production. The leftmost variable must always be expressed as terminals before proceeding since left recursion is not permitted. The production is resolved using the lookahead values LL(k) specified in the grammar. Most LL grammars use a value of LL(1), with localised values to resolve ambiguities.

In LR grammars, the converse takes place. The grammar is read from left to right; however the production rules are applied from right to left such that the rightmost variable must always be firstly resolved to a terminal. Like LL grammars, the default lookahead value is LR(k), however, localised values can be used to resolve ambiguities. LR grammars, are relatively easy to implement and can detect syntactic errors more readily than LL grammars, however LL grammars are more easily written.

2.3 Backus-Naur Form (BNF)

The Backus-Naur form of BNF is a system of representing context free grammars using metasyntax notation. Created by John Backus and extended by Peter Naur, it was firstly used as a means of describing the grammar of the Algorithmic Language (ALGOL). However BNF notation is now widely

used to construct the grammars of programming languages, communication protocols and instruction sets.

Backus-Naur notation (more commonly known as BNF or Backus-Naur Form) is a formal mathematical way to describe a language, which was developed by John Backus (and possibly Peter Naur as well) to describe the syntax of the Algol 60 programming language. (4)

The production rules of BNF define the syntax of the language being parsed. Each production defines a category in the language and the strings which are valid instances of each category along with the metasymbols used to define the structure of such the category.

The production is of the form:

$< non\ terminal >::= terminal \mid non\ termial$

The category is a non terminal left of the metasymbol "::=", while to the right is its production rule. The metasymbol "|" indicates an alternative terminal or non terminal and the productions can be increasingly nested until it is expressed by a sequence of language terminals only.

Such that:

$< non\ terminal >::= terminal \mid non\ terminal$

Is resolved to:

$< non\ terminal >::= terminal \mid (terminal \mid non\ terminal\ )$

and further:

$< non\ terminal >::= terminal \mid (terminal \mid (terminal\ ))$

BNF grammars typically consist of two main types: terminals and non terminals. Terminals are defined as strings of the language for which there exists no production rules and as such cannot be redefined with respect to the language. They are the alphabet of a language; defined by (3) as: "… a finite nonempty set of symbols … assumed to be indivisible". In standard programming languages including Java, a terminal could be language reserved words such as *do* and *for* or simply an integer. This is in contrast to language non terminals which cannot be consumed as tokens.

Non terminals are the symbols of the language for which there exists some production rule, such that they can be substituted with a series of terminal or non terminal symbols or a combination of both, to produce consumable language tokens by a lexical analyzer or parser. Non terminals, in BNF

representation, are to the left of the production rules indicating that they must be substituted with the symbols on the right.

The Extended Backus-Naur form (EBNF), provided for more flexible grammar constructs by allowing for simple operators to be added to the grammar definition. It introduced metasymbols to specify cardinality of the terminals and non terminals in a production rule. These new metasymbols include "?" representing zero or more occurrences of a symbol, "*" representing zero or one occurrences of a symbol or "+" representing one for more occurrences of a symbol. The metasymbol "[]" is used to indicate that a symbol is optional in the production.

2.4 JavaCC

Java Compiler Compiler (JavaCC) is a parser generator and lexical analyser. Grounded in the Java Language, JavaCC allows for the creation of parsers based on grammars defined using the Extended Backus-Naur Notation (EBNF). As the products of the parser generation process, JavaCC builds Java classes to implement lexical analysis, parsing, error and exception handling. This enables the compiler to inherit the desirable characteristics of the Java language such as platform independence, robustness and versatility.

Extended BNF notations are allowed in the JavaCC grammars. The support of eBNF (section 2.3) enables the creation of many easily read grammars, and to an extent negates the need for left-recursion grammar.

JavaCC EBNF Notation:

$String\ RelObjectName()$:
{}
{
$(< S\_QUOTED\_IDENTIFIER > \mid < S\_IDENTIFIER >)*$

{
}

JavaCC allows for the definition of context free grammars based on this general notation. The language of the grammar can be defined as tokens of the form:

$TOKEN$:
{
$< \#DIGIT: ["0" - "9"] >$
}

with the productions defined as simple Java methods of the form:

$$\boldsymbol{void}\ element() : \{\}$$
{
(
$< CONSTANT >$
)
}

This implementation of the production rules allows for the embedding of pure Java code at any level of the compiler. This additional code is directly inherited by the resulting classes and allow for more customized parsing, exception handling, user or machine interaction with the compiler. However, more importantly it enables the use of "companion tools" such as Java Trees (JJTree) to construct parse trees or binary abstract trees at runtime. Alternately embedded Java code can be used to create any desired output based on the parsed source.

Context free grammars are produced by the JavaCC compiler using the top-down or LL approach (see section 2.2). Using this approach, JavaCC grammars inherit the many advantages of LL grammars as described in section 2.2 Owing to Java's cross platform compatibility, a JavaCC generated parser may be expected to parse any commonly used language, an advantage of LL grammars. Another key advantage of the LL grammar in JavaCC, being its ability to pass attributes in both directions along the parse tree. This is a useful feature for embedded companion tools such as JJTree, its tree building pre-processor and JJDoc, its documentation generating tool. JJTree is used to build abstract syntax tree (see section 2.5.2) outputs from the parser. This is not used in this project as an IQL string is the required output format and any tree structure created would have to be deconstructed into IQL string. Therefore an IQL string will be created as the direct output of the parsing process.

JavaCC allows for the grammar and the lexical specification to be included in a single source file. This feature builds on the flexibility of JavaCC since a source file can contain at any level, regular expressions and non terminal strings interwoven with the grammar specification. The grammar by extension becomes much easier to develop, understood and maintained.

Another key feature of JavaCC is the ability of the programmer to define lookahead specifications at both the local and global scope. JavaCC compiles parsers which by default use a lookahead value of one. However as with many LL grammars, there often exist localized ambiguities where the default lookahead value is insufficient for the analyser to deduce the underlying terminal or non terminals. JavaCC provides for localized lookahead values (k), such that the default lookahead value is maintained while the locally defined values are used to resolve these ambiguities. Where (k)

represents the lookahead value or the number of tokens the parser needs to examine in advance before deciding on the current token.

The case-sensitive property of the Java Language can be exploited in creating grammar specifications which define strict case sensitive tokens. The tokens can be defined at both the local or global level by setting the Boolean property ignore_*case.* This can be used to enforce the case of keywords or for parsing case sensitive languages.

The debugging capabilities of JavaCC are more advanced than similar compiler compilers. Enabling debugging options such as the debug_lookahead and debug_parser provides for a detailed overview and analysis of the lexical analysis and parsing process. Debugging options allow for a JavaCC developer to trace and identify consumed tokens and evaluate the accuracy of the parser. This is particularly useful for resolving shift-shift ambiguities and determining localized lookahead values.

Error reporting is a vital requirement of parsers or lexical analysers. In this respect, JavaCC implements advanced error reporting superior to most compiler compilers or parser generators. As standard in any JavaCC parser generation, the TokenMgrError and ParseException error classes are produced. By default, these classes produce highly detailed reports with error location and expected tokens. These classes however, can be extended or directly modified to produce customized and more detailed error reports and diagnostic information.

2.5 Lexical Analysis and Parsing

2.5.1 Lexical Analysis

Lexical analysis is one of the main stages of modern language translation: see (5) who state that *Language translation has three phases: lexical analysis, parsing, and code generation.* Lexical analysis is usually the first stage of the translation process and involves the reading of the source code and producing, a formatted source according the grammar specification.

The lexical analyser may be summarised as

… the first phase of a compiler [whose] main task is to read the input characters and produce as [the] output, a sequence of tokens that the parser uses for syntax analysis. (6)

The lexical analyser also has the ability to execute certain secondary tasks at this early stage of the process and enhance the compilers' user interface.

The lexical analyser strips the input source code of new line characters, white space, blank lines, and tabs. In JavaCC these are defined as skip tokens at the beginning of the grammar specification in the form:

$SKIP$:
{
" "
|"\r"
|"\t"
|"\n"
}

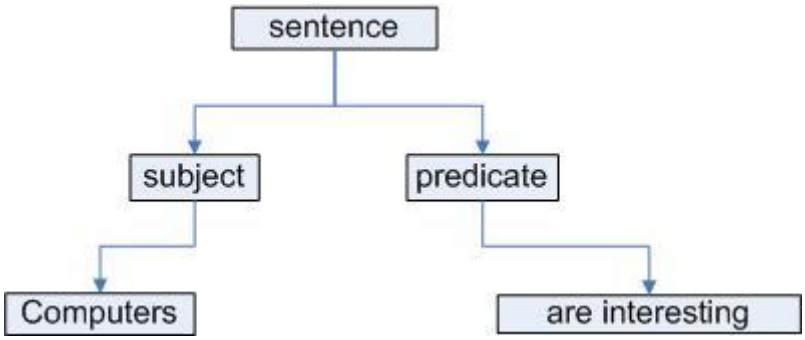
The analyser can maintain source code information such as line and character counts which can aid in providing the compiler with an accurate error report to the user. This may be manipulated by the developer to produce more specific error reports. The lexical analyser can also be used to implement pre-processor functions during this phase, provided it is a supported feature of the compiler.

Lexical analysis may take place as two separate phases, the scanning and lexical analysis phase or as a single harmonised phase. In the double phased approach, the scanning phase may be used for stripping the source code while the actual analysis takes place in the second phase. The double phased method allows for simplification of one of the phases such as removing non tokens in the scanning phase and creates a more efficient analyser. Platform compatibility of the compiler is also enhanced since platform dependent elements of the source code can be removed or substituted at the lexical phase. JavaCC gains no real benefit from this since its Java source is natively platform independent.

Lexical analysis however, is very restricted in its ability to detect compiler or runtime errors. While some stripping and substitution can take place, tokens such as strings or integers could have an infinite number of matches, and an analyser is incapable of filtering such tokens. There is also the limited ability to enforce agreement or select tokens based on the entire source structure. Such properties of lexical analysers are inherently a result of having a localized view of the source. (6)

2.5.2 Parsing

Parsing, which follows the lexical analysis phase, is used to determine if the strings given as input are valid in the languages specified by the parser. The parser creates an abstract syntax tree in the language of the grammar specified. An abstract syntax tree is an internal representation of a parsed input. It is a data structure that can be envisioned as an inverted tree, with the parsed data represented as the leaves.



2.2 - Parsed Sentence

In Figure 2.2, an English sentence, “Computers are interesting”, is represented in a parsed tree according to the grammar of the English language. The leaves contain the parsed data from the input sentence, while the root of the tree represents the input type being parsed. For this project, the output is required in the format of an IQL string and as such a parse tree is not created as a part of the translation process (see section 2.4).

As described in section 2.2, there exist primarily two forms of grammars; top-down or LL grammars and bottom-up or LR grammars. As such there exist two types of parsing, top-down and bottom-up parsing for the respective grammars. In the top-down approach, the root of the parse tree is first resolved and this proceeds down to the leaves, while the alternative approach of resolving from the leaves up the root is taken in the bottom-up approach. Most grammars, for development simplicity, employ a top-down approach as these grammars are more easily constructed. However as discussed in section 2.2, LR or bottom-up parsers are capable of handling a wider range of grammars but are more difficult to construct.

Context free grammars (see section 2.2) are generally used to define modern programming languages. These grammars are constructed by hand and then the corresponding parsers are generated using parser generators such as JavaCC (see section 2.2) rather than coding the parser

manually. This removes the need to create basic parser components such as a lexical analyser and parser compiler which are already readily available.

2.6 Database Systems

2.6.1 Database

Data persistence at some level has always been a requirement in programming. Early primitive systems stored data in ad-hoc flat files. These databases were designed for purpose, and with minimal interoperability, required a recompilation of client programs subsequent to any semantic or syntactic changes in the database. The lack of optimization, which was inherent in these databases, resulted from the challenges associated with their fragmentation and redundancy.

A database is a formally defined, organized collection of related data about the real world which can be accessed, shared and manipulated by users. In order to achieve this, modern databases are required to store data in a defined format, as prescribed by its related metadata and schema. The data must be stored independent of programs and users while remaining fully secured and accessible to both. The Database Management System (DBMS), the component responsible for this, further manages disk storage and optimization techniques for fast and efficient data management.

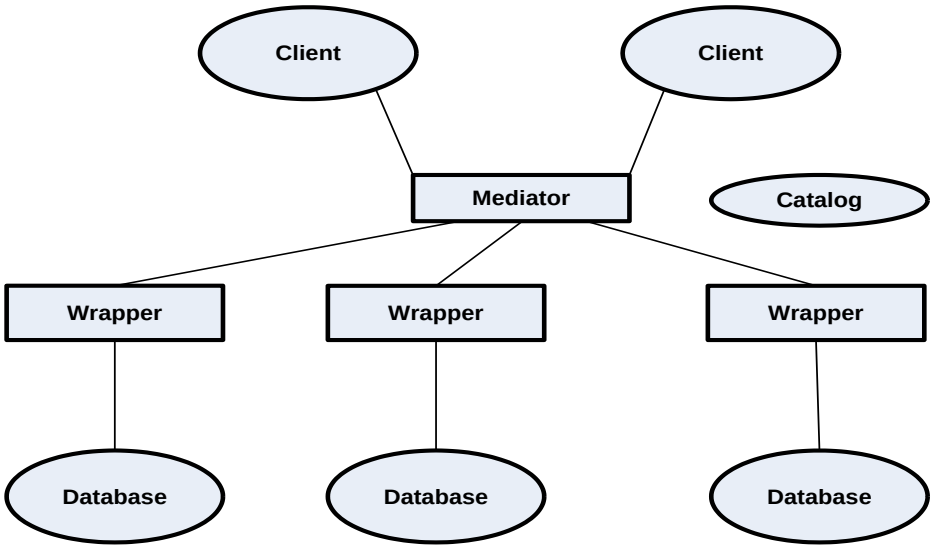
2.6.2 Relational Database

The Relational Database (RDBMS) exists as the most common database, with its basic object being the table or relation. Each table is structured with a number of rows (tuples) along with columns (attributes) with values describing each row. The relational model enforces constraints such as unique identifiers, required attributes and table referencing where a tuple N is related to some other tuple M in another table.

Through relational algebra, it supports the five basic operations of union, selection, difference, projection and product. However, most RDBMSs implement a higher level language such as SQL to allow users to express queries in a more meaningful manner. Common RDBMSs include Oracle, Microsoft SQL Server and Postgre SQL

2.7 Mediators

Heterogeneous database integration inherently poses problems of heterogeneity and autonomy. To create a common data model and schema, integration takes three main approaches: Warehouse integration, where all the data from the underlying local databases are imported into a central database from where queries are executed. *[This] emphasizes data translation, as opposed to query translation in mediator-based integration* (7). There is the Navigational Approach, a link-based approach where users are required to manually browse WebPages until the desired data is found. There is no relational model, and each link is simply a source satisfying the querying to different degrees. The mediator approach occupies the middle ground with respect to these two approaches.



2.3 - Mediator Based Integration

Mediator based integration, employs a series of transformation steps to map local data sources to a common data model (CDM). Database wrappers encapsulate the heterogeneity of the local databases. They translate queries, requests and the result set between the mediator and the sources. The CDM, also used in data warehouses, harmonises the varied heterogeneous data models such as SQL, Object-oriented and XML of the source databases. It is also used to express a global schema over which queries are written. The conceptual schemas are the point of translation between the local schemas and the CDM.

The Mediator …*concentrate[s] on query translation* (7) and does not influence local data storage or store data from the local sources. It is responsible for accepting and optimising global queries. The mediator however, maintains a catalogue of wrappers, schemas and data sources (see Figure 2.3). It passes sub-queries to the relevant wrappers, integrates the result sets and performs post query functions such as grouping and sorting. The mediator cannot resolve source specific instructions or queries aimed at specific sources.

2.7.1 Global-as-View

Integrating the databases involves using one of two main methods of mapping the schemas: the Global as View approach (GAV) or the Local as View approach (LAV). In the GAV approach, the global schema is defined as constructs over each local schema and a global query is simply a query over the local schemas (8). The GAV approach thus facilitates querying across the local schemas, however, it makes for a more difficult system to maintain since the addition or removal of sources requires a redefinition of the global schema with respect to the current sources. This renders the GAV approach to meditation less optimal for a large and diverse integration but recommended for a mediator with a stable set of sources.

2.7.2 Local-as-View

In The LAV approach, however, the local export schemas are defined with respect to the global schema. This improves the extensibility across large or ad-hoc databases since the onus is on the local sources to create the export schemas as described by the global schema. Sources can be readily added or removed with little or no changes to the system other than the changes in the underlying data. However, this simplicity in integrating schemas makes the processing of queries a more difficult process since the export schemas only represent the data in terms of the knowledge of the global schema. This often represents only a partial view of all the underlying data and extracting all the information from the sources becomes a complex task because one has to answer queries with incomplete information (9).

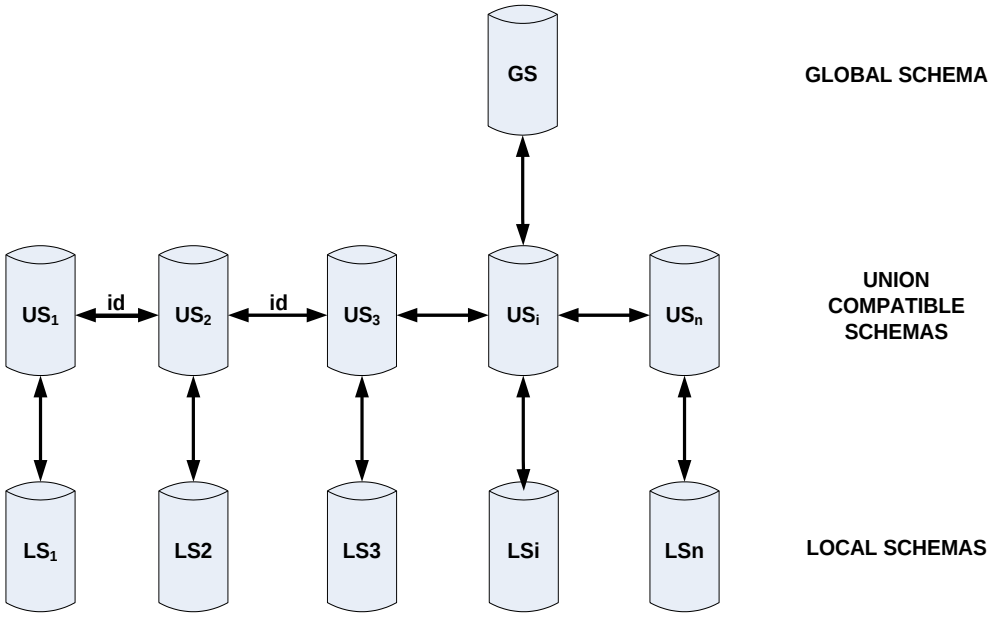
The basic problem of LAV is that in order to answer a query on the global schema, the LAV rules (defined as views of local schema constructs over the global schema) need to be rewritten somehow and this problem is difficult.

The disadvantages of both methods have prompted the suggestion for a combination of both views. (10) The GAV and LAV are combined to create the both as view (BAV) integration as used in the AutoMed mediator to create a reliable and scalable integration system while maintaining speed and usability

2.8 AutoMed Heterogeneous Data Integration System

AutoMed is a mediator developed by Birkbeck and Imperial Colleges. Traditional heterogeneous integration involved the use of a Common Data Model (CDM) using relational, object –oriented or graph based approaches. The main drawback with this approach is the existence of a single data model and by extension a single global schema.

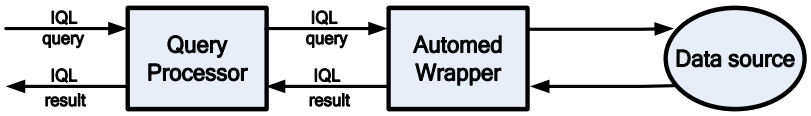
The AutoMed approach implements a low level Hypergraph-based Data Model (HDM) to define the higher level languages of Object Oriented, Relational or XML using the AutoMed Model Definitions Repository (MDR) (11). This removed the limits of a single Data Model, but rather permits the use of any model defined within the MDR. The HDM consists of nodes and edges in a Hypergraph such that each higher level language is defined as a combination of these.



2.4 - AutoMed Integration Model

AutoMed provides primitive transformations which can be applied to the constructs of a schema expressed in any modelling language, M. Through a series of transformations or composite transformation, each local schema, LSi, in Figure 2.4, is transformed into a union compatible schema USi where $LS1 \rightarrow US1, LS2 \rightarrow US2, LSn \rightarrow USn$.

The resulting union schemas $US1$ … USn are syntactically identical. The transformation pathway is expressed in terms of the changes in the constructs of the schemas where $USi \rightarrow US(i+1)$ consist of the transformation steps:$id(USi : C, US(i+1) : C)$. Where the id is an additional type of primitive transformation and the notation $USi : C$ is used to denote the construct *C* appearing in schema USi.This approach constitutes AutoMed's both-as-view or BAV approach to schema integration. The BAV approach is much superior to the GAV and the LAV approaches since it is possible to define the global schema in terms of the source local schemas as well as define the source schemas as views over the global schema. BAV represents a more expressive integration approach since mappings can be expressed in both directions with no limits on the number of source schemas that can be integrated (12).



2.5 - AutoMed Global Query Processor (GQP)

AutoMed includes the AutoMed Global Query Processor (AQP) to evaluate mediator queries across the global schemas. The AQP accepts queries which are reformulated, optimised and annotated before being submitted to the AutoMed wrapper objects for evaluation against the local schemas.

AutoMed queries natively are expressed in the Intermediate Query Language (IQL), a lower level functional language. However this maybe modified to accept higher level languages such as SQL and XQuery (1).

3 APPROACH

Implementing an SQL to IQL component for the AutoMed toolkit is potentially a complex task requiring a lot of research, planning and implementation. The chosen approach, must be rigorous in its reliability, accuracy and efficiency while remaining flexible towards customisation and extensibility. The time constraints on the project require that any chosen approach also be modular to enable usability of parts the solution while allowing for future completion. It should also be a documented approach which would require minimal new knowledge to implement or modify.

The first possible approach examined, was the creation of a translator component in Java to accept and convert SQL to IQL. With this approach the solution would be written in the same language as AutoMed and could be simply added to the existing libraries. Implementation would require minimal new knowledge with respect to the languages and would be relatively easier to maintain on completion. Along with the inherited benefits of Java, the component would also benefit from being written in the native language of the AutoMed toolkit. A disadvantage of this approach would be the considerable programming effort needed to implement a translator for all possible combinations of SQL statements. Achieving this would also result in a relatively complex and slow component which would be challenging to maintain and extend. This defeats one of the main objectives of an approach: to have a minimum implementation and maintenance time factor. To avoid this, a parser generator could be used to generate a translator from a grammar specification.

Bison as a GNU parser generator was examined as the second possible approach. It is maintained by Paul Eggert and Akim Demaille and available from http://www.gnu.org/software/bison.Creating a parser specification would resolve the time constraint factor since it requires relatively less effort than specifying a translator in raw code. Bison grammars are specified in BNF notation to generate LALR (1) C or C++ parsers for varying language complexity. Bison requires that lexical analysis be done by a third party tool such as The Fast Lexical Analyser (Flex) available from http://flex.sourceforge.net. The reliability and speed of the C-based languages allows Bison to create superior parsers which, with recompilation can be executed on any platform. As a disadvantage, the use of C-based parsers would limit integration into AutoMed which is a Java based program. The resulting parser would have to be re-compiled for all possible platforms. Adding an external lexical analyser also creates an extra layer of complexity. This would reduce compatibility and increase maintenance requirements, defeating the objectives of the chosen approach. The LALR (1) parsers generated, are less efficient in resolving ambiguities than the LL grammars used in JavaCC (2.4), since in comparison all tokens must be consumed before a choice is made (2.2). Although Bison parsers

are also created in BNF notation, this falls short in specifying choices, sequences, loops and number of terminals. As a result the grammars are more complex, difficult to understand and maintain. This further limits the benefits of this approach which requires more advanced and compatible parser generator.

Building a parser generator using JavaCC from https://javacc.dev.java.net/ or Java Cup from http://www2.cs.tum.edu/projects/cup was taken as the most suitable approach. As discussed in 2.4, the advantages of JavaCC make it a suitable tool for generating parsers. They key difference between JavaCC and Java Cup being that JavaCC supports internationalization. This advantage is crucial if the resulting parser is to be used for languages outside of the English language. Unlike pure Java, and like Bison, JavaCC simplifies creating the translator through a relatively generic specification. This obeys the time constraints of with respect to the implementation and maintenance of any complex code. The Java background of JavaCC enhances compatibly with the toolkit and would require minimal new knowledge. The resulting parser could be as simple as needed and readily extended beyond the original specification. These advantages make JavaCC the most suitable approach for implementation.

4 STRUCTURED QUERY LANGUAGE

4.1 SQL

Structured Query Language (SQL) was developed by IBM in the mid 1970's as means of querying relational databases. SQL, over a short period of time, had become the de facto query language for relational database management systems. It subsequently gained acceptance by the American National Standards Institute and the International Organization for Standardization in 1986 and 1987 respectively. Further revisions were adopted in 1989, 1992, 1999, 2003 and 2006. This scope of this paper is restricted to ANSI compliant SQL 1992.

As a declarative language, SQL remains confined to "what" instead of "how" when querying databases. The advantage being simplicity as the user defines the conditions for the data set to be returned, but almost always never provides any instructions on how the data is to be retrieved. This task is undertaken by the database engine and varies with the Database Management Software. SQL can be subdivided into two main parts; the Data Definition Language (DDL), and the Data Manipulation Language (DML) for database management.

The DDL is used to define the database and its constituent objects. It is not involved in data manipulation and includes the commands: $CREATE, USE, ALTER, and\ DROP$. DDL is restricted to database administrators is used to manipulate the database objects and schemas and to set constraints, relationships, indices and namespaces.

The DML is the more commonly used sub-language and is routinely used by all database users for inserting, retrieving, manipulating data. The DML includes the commands: $INSERT, UPDATE, DELETE$ and the most frequently used $SELECT$. This project will be focused on the translation of the $SELECT$ statement subset of the DML sub-language.

4.2 Select Statement

The select statement comprises of three distinct parts: select, from and where. These parts can be directly linked to the structure of relational algebra constructs, where the select clause is the implementation of the projection operation. The from clause is the implementation of the Cartesian product operation. The where clause corresponds with the selection predicate of the relational algebra involving the attributes of the relations expressed in the where clause (13).

For two collections M and N, both having attributes $\{a, b, c\}$ a typical SQL query, selecting $\{a, b, c\}$ from M and N joining on attribute $\{c\}$ would be expressed as:

$SELECT\ a, b, c$	Attributes- Select Clause
$FROM\ M, N$	Relation – From Clause
$WHERE\ M.c = N.c;$	Predicate – Where Clause

The from clause is firstly evaluated, followed by the where clause and finally the select clause.

4.2.1 Select Clause

The select clause contains the list of attributes or columns that will be returned from the underlying query. The standard select statement implies a $select\ all$ operation aimed at reducing the cost overheads of eliminating duplicates from result set. The $distinct$ function can be optionally applied to remove duplicate tuples.

It may also contain aggregation and grouping functions that are applied to the attributes before projection. These include numeric operators +, -, * and / as well as aggregators: sum, avg, min, max and count. SQL allows for aggregation to be applied between attributes or a combination of attributes and real numbers (e.g. 1.2, 2.5) or integers (e.g. 1, 2, and 3). A query, returning the average of the sum of the attributes {a, b} of all tuples in M may be expressed as:

$SELECT\ avg(a + b)$
$FROM\ M;$

With the exception of nested queries, SQL does not require fully qualified attributes, however where attributes are common across multiple relations, this maybe done to reduce any potential ambiguities. The select statement above could be expressed fully qualified as:

$SELECT\ avg(M.a + M.b)$
$FROM\ M;$
In order to return all the attributes, the asterisk symbol could be substituted for attributes names indicating that all the attributes resulting from the from clause be returned by the query. This is expressed as:

$SELECTM.*$
$FROM\ M;$

4.2.2 From Clause

The from clause of the select statement contains the relations to be scanned while evaluating the underlying query expression. An example is as follows:

$$FROM\ M, N$$

When expressed as a statement with only select and from clauses, the from clause is evaluated as the cartesian product of all the relations. Further filtering can be achieved by explicitly defining joins or filtering in the where clause. Nested queries are fully supported in the from clause and like the relations, can be manipulated within the query.

4.2.3 Where Clause

The where clause of the select statement is the filtering clause allowing a query to return those tuples which are of interest to the user. It is expressed as follows:

$$WHERE\ M.a\ >\ 5\ and\ N.b < 10$$

This clause follows the from clause and consists of filtering conditions using the logical connectives (and, not and or) as well as string and numeric comparison operators ($<, >, =, >=, <=, <>$). Both string and numeric expressions can be evaluated as well as special types such as dates (13). Nested queries are fully supported in the where clause and like the relations, can be manipulated within the query.

4.2.4 Aggregation

SQL provides for five built in aggregation functions: avg, min, max, sum and count. Aggregation is supported across different data types including strings and numeric types. However, the functions avg and sum can only operate on numeric data types. A query, returning the average of the sum of the attributes {a, b} of all tuples in M may be expressed as:

$$SELECT\ avg(M.a + M.b)$$
$$FROM\ M;$$

This query produces a result set containing a single value for the average value. The aggregation function is more commonly used with group by functions to produce averages based on a common grouping of tuples instead of the entire relation.

The retention of duplicate tuples will affect the accuracy of any aggregation function executed. In relations, this maybe done on a commonly repeated attribute, such as a sales item attribute in a sales table, the removal of any sales item tuple based on this attribute could result in an inaccurate aggregate returned. Duplicates however can be eliminated using the distinct function when needed. The distinct function, while legal by the SQL BNF, effects no changes to the result of max and min and is not permitted with the count (*) function.

Where grouping is used along with an aggregation, further filtering can be accomplished by means of the $having$ clause. This clause allows a higher level of filtering based on the values returned and unlike the where clause, filters on groups of tuples rather than individually. The $having$ clause is outside the scope of this implementation.

4.2.5 Nested Queries

SQL provides full support for the arbitrary nesting of sub-queries. A sub-query is a select-from-where expression that is nested within another query (13). The nested query may be used in the from clause in instead of a schema relation. The result set produced from the execution of the nested query provides the tuples for the outer query. A query with nesting in the from clause may be expressed as:

$SELECT\ T.a, T.b$
$FROM$
(
$SELECT\ M.a, \quad M.b$
$FROM\ M$
$WHERE\ M.a > 10$
) $as\ T$; ~

When nested in the $from$ clause as a derived relation, the nested query must be renamed using the as clause and the attributes may also be renamed to avoid ambiguities in the result set. This renaming can occur within the derived query or in the select clause of the outer query.

SQL further allows for testing set membership using the in or $not\ in$ functions. This is typically employed in the $where$ clause for filtering. An example of testing for test membership is:

$SELECT\ a, b$
$FROM\ N$
$WHERE\ a\ NOT\ IN$
(
$SELECT\ a$
$FROM\ M$
$WHERE\ a > 10$
);

Where, for each tuple in M, its attribute is evaluated for membership in the set returned from the sub-query. The sub-query could return a single aggregate value and as such the where clause could evaluate this using the standard comparison operators. Such an example would be:

$SELECT\ a, b$
$FROM\ N$
$WHERE\ a >$
(
$SELECT\ MAX(a)$
$FROM\ M$
);

4.2.6 Set Operators

SQL supports full set operations including union, intersection, union all and minus. SQL requires that the relations participating in the operation must be compatible. They must have the same number and types of attributes.

The union and union all operators append both the results of both queries to produce a result set. The union query by default eliminates duplicate tuples, however union all may be used when needed to retain all tuples. A simple union query on all the attributes of M and N maybe expressed as:

$(SELECT\ *$
$FROM\ N)$
$UNION$
$(SELECT\ *$

$FROM\ M$);

All the set operators can be similarly used. They can both be nested or contain full SQL nested queries.

These represent the basic structure of SQL statements. More complex statements may be constructed and are compared in section 6 below and tested in section 8 below.

5 INTERMEDIATE QUERY LANGUAGE

5.1 IQL Structure

The AutoMed Intermediate Query Language (IQL) is a typed, comprehension-based functional query language (1). IQL provides for a more expressive query language than higher level languages which makes it ideal for querying mediators systems such as AutoMed.

Data in IQL can be expressed as strings in single quotes (e.g. 'Jamie Walters'); real numbers such as 10.25; Boolean values (True, False) and integers (1, 2, 3…). A data collection is represented as tuples, lists, sets or bags which can be nested or manipulated using the built-in functions and operators as well as custom user defined functions (14).

As a functional language, IQL contains the basic built-in operators and functions (e.g. +, -, *, and, not). These operators can be expressed in prefix form where they are enclosed in brackets, (-) 10 7, or in the infix form, 10 – 7 where both expressions are equivalent and yield the same results (14).

As a query language, it can be used to express relational algebra constructs as is expressed in the higher level languages such as SQL. These include operators, selection, projection and joins. For two lists M and N consisting of tuples of the form {a, b, c}, an example set operation:

Union All of M and N is expressed as:

$M\ union\ N$

Or

$union\ M\ N$

The same applies for the remaining set operators: intersect (intersect), union (++) and minus (--).

Using IQL, a comprehension can be used to express selection or projection of tuples in a collection. For the collection M as stated above, a projection returning all components {b, c} from all the tuples in collection M may be expressed as:

$[\{b, c\} \mid \{a, b, c\} \leftarrow M]$

The comprehension can be further used to express a selection of the tuples in M based on some specified filtering restrictions. For the collection M as stated above, a projection returning the

components {b, c} from all the tuples in collection M where the component c has a value greater than ‘6 ‘ may be expressed as:

$[\{b, c\}|\{a, b, c\}\leftarrow M;\ c > 6]$

Where relational data exists across several collections, the IQL comprehension syntax fully supports the use of Cartesian products and joins to project a single collection of tuples over components from all joined tables. A join of collection M and N on their c components, with a projection on {a, b}, maybe expressed in IQL comprehension as:

$[\{a1, b1\} \mid \{a1, b1, c1\} \leftarrow M;\ \{a2, b2, c2\} \leftarrow N;\ c1 = c2]$

The built in functions *and*, *or* and *not* maybe used to create more complex joins using the Boolean result of a combination of sub-filters in the join and selection clause. A more complex join of M and N could be on both their b and c components:

$[\{a1, b1\} |\{a1, b1, c1\}\leftarrow M;\ \{a2, b2, c2\}\leftarrow N;\ ((b1 = b2) and\ (c1 = c2))]$

Collections in IQL can be successively nested to build more detailed and useful result sets. This permits IQL to support the nesting of any query that return a valid collection of tuples which can then become the input for the outer query. The above query nested and filtered by the tuples with b = 10 could be expressed in IQL comprehension as:

$$[\{a1, b1\}|\quad \{a, b\} \leftarrow \left[\{a, b\}\middle|\{a1, b1, c1\}\leftarrow M;\ \{a2, b2, c2\}\leftarrow N;\ \left((b1 = b2) and\ (c1 = c2)\right)\right];\ b = 10]$$

The IQL comprehension supports grouping and aggregation operations (14). These include: count, sort, distinct, max, min, avg, sum and group. To count all the tuples in collection M:

$count\ M$

To find the maximum value of component {a1} of all the tuples in M:

$max\ [a1 \mid \{a1, b1, c1\} \leftarrow M]$

A comprehension, grouping on the components {a1, c1} in M and returning all components could be expressed as:

$group\ [\{\{a1, c1\}, b1\} \mid \{a1, b1, c1\} \leftarrow M]$

To apply both grouping and aggregation to a collection, the gc function is used along with the aggregation function. For example, a query that groups collection M by its components {a1, b1} and return the sum of {c1} across the resulting tuples would be expressed as:

$gc\ sum\ [\{\{a1, b1\}, cb1\} \mid \{a1, b1, c1\} \leftarrow M]$

5.2 IQL in AutoMed

In AutoMed, IQL is used to express the add, contract, extend and delete transformations used to create the global schema. However, it is also the language used to express queries over the any of the schemas defined within the AutoMed Schema Transformation Repository (2.8).

An example of an AutoMed IQL query is:

$(distinct\ [\{cn, u\} \mid \{c, cn\} \leftarrow \text{‹‹}course, coursename\text{››};\ \{c, u\} \leftarrow \text{‹‹}course, programme\text{››};\ c > 10)]$

This statement returns a unique list of coursenames and programmes where the value of the key of each tuple is greater than 10.

The comprehension consists of primarily two parts

$\{cn, u\} \mid$	Head
$\{c, cn\} \leftarrow \text{‹‹}course, coursename\text{››};\ \{c, u\} \leftarrow \text{‹‹}course, programme\text{››};\ c > 10)]$	Body

The head of the IQL comprehension is equivalent to the select clause in a standard SQL query. It contains the attributes to be returned from the resulting result list along with any functions and aggregation to be done on the columns or the result set.

The body of the comprehension is the section of the IQL query contained after the "|" character. The body comprises of the generators and the filters:

$\{c, cn\} \leftarrow \langle\langle course, coursename \rangle\rangle$; $\{c, u\} \leftarrow \langle\langle course, programme \rangle\rangle$;	Generators
$c > 10$	Filters

Each generator consists of a schema construct and the pattern of the attributes being returned:

$\{c, cn\}$	Pattern
$\langle\langle course, coursename \rangle\rangle$;	Scheme

The generator returns a list of tuples with two attributes: the key of the schema construct and the attribute. This limits each generator to returning a single attribute, requiring a new generator for each attribute to be returned from a table. There is no requirement for the variable names of the attributes in the pattern to match the names in the schema. The relation between both is resolved by the position of each variable name in the pattern. Variable names are local to a schema or a comprehension and any external reference is resolved using the variable patterns.

The filter in an IQL query equates to the where clause of SQL statements. The filter clause is used for refining selection or creating joins. The structure of filter is discussed in more detail in section 6.2. The AutoMed implementation supports the arbitrary nesting of queries, the use of functions and the supported relational algebra constructs of IQL. A comparison of the languages and the translation issues involved is discussed in the following chapter. The SQL must be translated accurately by adhering to the definition of an IQL comprehension.

6 TRANSLATING SQL TO IQL

The AutoMed Intermediate Query Language (IQL) is a typed, comprehension-based functional query language. As a functional language, IQL provides for a more expressive language for querying which makes it ideal for use in mediator environments such as AutoMed (1). IQL is a common language which can be readily translated to and from higher level languages such as SQL and XQuery (1). Previous work has been done in this area however; there is need for an SQL translator to make AutoMed more accessible to users

6.1 Data Types

IQL like SQL supports integers (e.g. 0, 1, and 2) and Boolean types (e.g. True | False) as well as floats (e.g. 1.5, 0.012); implemented as Java primitives. It also supports strings enclosed in single quotes (‘e.g. Jamie’) and date time objects (e.g. dt ‘2007-08-14’), both of which are represented as Java String objects. Other data types supported include lists (e.g. 1, A, C), bags and sets

6.2 Relational Calculus

Like the higher level languages such as SQL, IQL provides full support for relational calculus. For simplicity and to aid in formatting, some columns have not been expressed in this section in a fully qualified format. However, the translator requires fully qualified SQL statements and produces fully qualified IQL queries. The following SQL statement:

$SELECT\ id, \mathrm{n}ame$
$FROM\ staff$
$WHERE\ id\ >\ 5;$

Would be expressed in IQL as:

$[\{id, name\}|\quad \{staffPK, name\} \leftarrow \langle\langle staff, name\rangle\rangle;$
$\{staffPK, id\} \leftarrow \langle\langle staff, id\rangle\rangle; id > 5]$

The simple select statement in IQL consist of a head: $(\{ID\ Name\}|)$ This equates to the $SELECT$ clause the SQL query statement and contains the attribute of the tuples to be returned by the query. The second part of the query is the

comprehension: $(\{staffPK, name\} \leftarrow \langle\langle staff, name\rangle\rangle;\ \{staffPK, id\} \leftarrow \langle\langle staff, id\rangle\rangle;)$ which equates to the $FROM$ clause and the filter: $(id > 5])$ which equates to the $WHERE$ clause of the query statement.

IQL supports all the functions of relational algebra which are also supported by the higher level query languages. It supports the set operators $UNION$, $UNION\ ALL$, $INTERSECT$ and $DIFFERENCE$.

For example the following SQL UNION ALL statement:

$SELECT\ id, name$
$FROM\ staff$
$WHERE\ id\ >\ 5;$

$UNION\ ALL$

$SELECT\ id, name$
$FROM\ student$;
$WHERE\ id\ >\ 5;$

would be expressed in IQL as:

$$
\begin{array}{ll}
([\{id, name\}| & \{staffPK, \mathrm{name}\} \leftarrow \langle\langle staff, \mathrm{name}\rangle\rangle; \\
 & \{staffPK, id\} \leftarrow \langle\langle staff, id\rangle\rangle; id > 5]) \\
 & ++ \\
([\{\mathrm{id}, name\}| & \{studentPK, name\} \leftarrow \langle\langle student, \mathrm{name}\rangle\rangle; \\
 & \{studentPK, id\} \leftarrow \langle\langle student, id\rangle\rangle; id > 5])
\end{array}
$$

The second list or bag is simply appended to the first maintaining duplicate tuples. The SQL $UNION$ has an equivalent operator in IQL. Where a query such as $UNION\ ALL$ returns duplicates, the DISTINCT function $\left(distinct\ (set1\ ++\ set2)\right)$ can be used to remove duplicates and create the equivalent UNION result list. Alternatively the $UNION$ operator in IQL ($UNION\ set1\ set2$) creates an SQL $UNION$ equivalent result set.

The SQL INTERSECT statement:

$SELECT\ id, name$
$FROM\ staff$
$WHERE\ id\ >\ 5;$

$INTERSECT$

$SELECT\ id, name$
$FROM\ student$
$WHERE\ id\ >\ 5;$

Is represented in IQL as

$$intersect$$
$$[\{id, \mathrm{name}\} \mid \{staffPK, name\}" \leftarrow" \ \langle\langle staff, name\rangle\rangle;$$
$$\{staffPK, id\} \leftarrow \langle\langle staff, id\rangle\rangle; id > 5]$$
$$\{studentPK, name\}" \leftarrow" \ \langle\langle student, name\rangle\rangle;$$
$$\{studentPK, id\}" \leftarrow" \ \langle\langle student, id\rangle\rangle; id > 5]$$

The intersect operator produces a similar result set as its SQL equivalent, the only difference being in the construct of the query. Other set operators can be easily translated into an IQL equivalent construct. The complete translation functions for the SQL union operators are listed in table

6-1 - SQL to IQL Set Operators

.

SQL	IQL
UNION	union
UNION ALL	++
INTERSECTION	intersection
MINUS	--

6-1 - SQL to IQL Set Operators

Like SQL, IQL can be used to express projection over the tuples of a list or table. For any result set or table T, containing columns $\{a, b, c\}$ and SQL projection of $\{a, c\}$ across the tuples, would be expressed as:

$SELECT\ a, c$
$FROM\ t;$

In IQL this is represented as:

$[\{a, c\} \mid \{a, b, c\} \leftarrow T]$

For specifying components of the tuples in both languages, translation is relatively easy in both directions. However, one limitation of the lower-level IQL language is the inability to simply project

on all the components of the tuple with no knowledge of the underlying schema. This is an option frequently used in SQL and is expressed as:

$SELECT\ *$
$FROM\ T;$

The equivalent IQL projection would require knowledge of the components of the tuple and expressed more precisely as:

$[\{a, b, c\} \mid \{a, b, c\} \leftarrow T]$

Translating this SQL command would require pre-fetching of the IQL schema to resolve the tuple components. However this would only be possible where a projection is being made across a known table or view and require a complex task of maintaining the flow of tuples up the query structure from the lowest nested table. Projection is usually the done over some underlying selection function.

Selection in SQL is represented in the body of the query statement as the FROM and WHERE clauses combined. For example in the SQL statement:

$SELECT\ id, name$
$FROM\ staff$
$WHERE\ id\ >\ 5;$

This selection is interpreted as returning the *id* and *name* components of all tuples in the relation staff, where the id component is greater than 5. IQL similarly supports selection in comprehension as:

$[\{id, name\} \mid \quad staffPK, name\}" \leftarrow "\ ‹‹staff, name››;$
$\{staffPK, id\} \leftarrow ‹‹staff, id››; id > 5]$

This is a relatively straightforward translation of the select statement into an IQL comprehension. The comparison operators (e.g. <>) of IQL differ from that of standard SQL (e.g. <>, ! =) and would require translation and decomposition of the $WHERE$ clause to create an IQL compatible filter.

IQL supports the relational algebra Cartesian product and joins. Cartesian products are cost intensive database queries where each row of the first table is joined with every row of the second table to create all possible combinations of the two tables. An example of the cartesian product of two tables in SQL is represented as:

$SELECT\ \ name, id$

$FROM\ staff, student;$

In IQL this is represented as:

$[\{id, name\} |\ \ \{xstaffPK, name\}" \leftarrow "\ \langle\langle staff, name\rangle\rangle;$
$\{xstudentPK, id\}" \leftarrow "\ \langle\langle student, id\rangle\rangle]$

A standard join is expressed as

$SELECT\ \ staff.name, student.id$
$FROM\ staff, student$
$WHERE\ staff.id = \ student.id;$

Is expressed in IQL as:

$[\{id, name\}|\ \ \{staffPK, name\} \leftarrow \langle\langle staff, name\rangle\rangle;$
$\{studentPK, pid\} \leftarrow \langle\langle student, id\rangle\rangle;$
$\{staffPK, sid\} \leftarrow \langle\langle staff, id\rangle\rangle;$
$pid = sid]$

This query joins the staff and student table over the id column and projection on the $\{id, name\}$ tuple components. This is an extended method of creating joins on tables in IQL as its variable unification support allows for simple, compact and optimized query writing. Duplicate variable names are allowed in comprehensions, where a duplicate variable indicates a join on table using the tuple components represented by the duplicate variables. The IQL query processor will rename the variables, eliminating duplicates and creating the fully qualified comprehension. The IQL query:

$[\{id\ name\}|\ \ \{staffPK, name\} \leftarrow \langle\langle staff, name\rangle\rangle;$
$\{studentPK, pid\} \leftarrow \langle\langle student, id\rangle\rangle;$
$\{staffPK, sid\} \leftarrow \langle\langle staff, id\rangle\rangle$
$;\ pid = id]$

Could be written by a user as:

$[\{id, name\}|\ \ \{staffPK, name\} \leftarrow \langle\langle staff, name\rangle\rangle;$
$\{studentPK, id\} \leftarrow \langle\langle student, id\rangle\rangle;$
$\{staffPK, id\} \leftarrow \langle\langle staff, id\rangle\rangle]$

This would be resolved by IQL to the longer formed query which is then passed on to the query processor.

One of the main differences in the translation of the where clause is the set membership function. This common SQL function is used to test the membership of tuples to a specified collection which could be another derived table. In SQL this is represented as:

$WHERE\ x\ IN\ R$;

The IN function does not exist in IQL and the alternative *member* function or its negated form is used. This query would be expressed in IQL simply as:

$member\ R\ x$

The $NOT\ IN$ SQL function would be expressed as:

$not\ (member\ R\ x)$

The complete list of comparison operators as translated in the where clause is listed on table 6-2 - SQL to IQL Comparison Operators

.

>	>
<	<
<>	<>
=	=
in	member
not in	not(member)
like	like
not	not

6-2 - SQL to IQL Comparison Operators

Like SQL, IQL can be arbitrarily nested to include ad hoc derived tables in both the *FROM* and *WHERE* clauses. The key difference in IQL being that unlike, SQL, the outer query must accept all the columns returned by the nested query. This as IQL implements a list collection to return the results, and the names of variables or columns are local to a comprehension and cannot be seen

by the outer query. The order of the columns must be maintained so that the columns are correctly assigned.

The SQL query:

$SELECT\ \ people.name$
$FROM$
$(SELECT\ \ staff.name, student.id$
$FROM\ staff, students$
$WHERE\ staff.id = \ student.id) AS\ people;$

Would be represented in IQL as:

$$[\,\{\,people\_name\ \}\,|\quad \{people\_name, people\_id\} \leftarrow [\,\{\,staff\_name, student\_id\ \}\,|\quad \{staffPK, staff\_name\} \leftarrow \langle\langle staff, name\rangle\rangle; \{studentPK, student\_id\} \leftarrow \langle\langle student, id\rangle\rangle; staff\_id\ =\ student\_id\,]$$

Unlike the SQL statement, both attributes from the derived IQL comprehension $\{People.name, People.id\}$ are referenced in the outer query. This difference would require additional translation rules to maintain, track and reference all nested queries.

Grouping and aggregation remains relatively similar to SQL queries. IQL allows for grouping or aggregation on single or multiple columns, with a change in the syntax. While an SQL statement allows grouping over aggregation over multiple columns in a single statement, IQL only allows a single aggregation in each comprehension and multiple aggregation is represented as a series of nested comprehensions where the outer comprehension returns each a list of all the aggregation as a single result set. This makes it more complicated to manage multiple aggregations the SQL statement:

$SELECT\ max(people.id), min(people.name)$
$FROM\ people;$

Would be represented in IQL as:

$$[\,\{\,people\_id, people\_name\ \}\,| \quad \{\,people\_id\,\} \leftarrow [\,max\,\{\,people\_id\,\}\,|\quad \{peoplePK, people\_id\} \leftarrow \langle\langle people, id\rangle\rangle; \{peoplePK, people\_name\} \leftarrow \langle\langle people, name\rangle\rangle\,]; \{\,people\_name\,\} \leftarrow$$

$$[\ min\ \{\ people\_name\ \}\ |\quad \{peoplePK, people\_id\} \leftarrow\ \langle\langle people, id\rangle\rangle;$$
$$\{peoplePK, people\_name\} \leftarrow\ \langle\langle people, name\rangle\rangle\ ];\ ]$$

Group by functions are relatively similar between IQL and SQL.

The SQL:

$SELECT\ people.id, people.name$
$FROM\ people$
$GROUP\ BY\ people.id;$

Would be translated as:

$group\ [\{people\_id,$
$\{people\_name\ \}\}\ \ |\qquad \{peoplePK, people\_id\}\ \leftarrow \langle\langle people, id\rangle\rangle;$

$$\{peoplePK, people\_name\}\ \leftarrow\ \langle\langle people, name\rangle\rangle\ ]$$

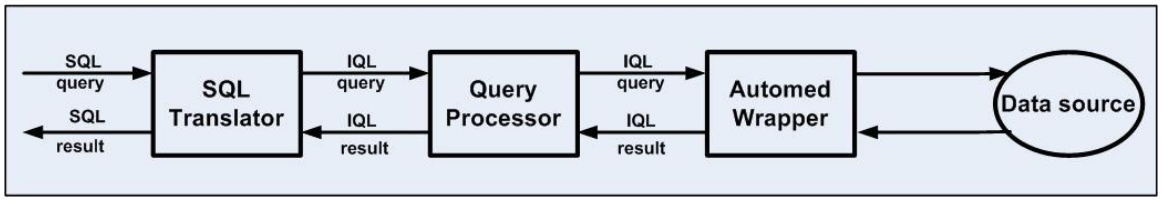
The group by clause is moved to the head of the comprehension and the grouping co column separated using parenthesis and commas. This is straightforward and requires simple rules, while making no changes to the body of the comprehension.

Having identified the possible areas for attention in translating from IQL to SQL, the translator can be implemented based in these defined rules. The approach provides a suitable development environment and language to create an accurate and useable solution.

7 IMPLEMENTATION

The approach chosen for the implementation was the creation of a parser generator using the JavaCC tool. This required the specification of an LALR grammar which would then be compiled into a Java class and integrated into the AutoMed toolkit. The implementation required several steps and methods. The translator is platform independent; however it requires Java virtual machine.

7.1 Implemented Solution



7.1 - The SQL Translator in AutoMed

Using the approach chosen, an SQL translator would be integrated into the AutoMed toolkit as shown in figure 7.1. The translator, created using Java would accept the SQL queries, translate them to the equivalent IQL representation and submit them to the AutoMed Query Processor. The AQP would return the IQL formatted query results to the translator. The results would then be formatted by the translator and returned to the user.

7.2 Tools

The JavaCC grammar was specified using the Eclipse IDE tool (http://www.eclipse.org) and the Eclipse JavaCC plug-in (http://sourceforge.net/projects/eclipse-javacc). The plug-in allows for the specification of grammars in an interactive and efficient environment. The grammars can be created as classes in a Java project and compiled and tested incrementally. The compilers used for the project were JavaCC 4.0 for compiling the grammars and Java SE 1.4.2 for compiling the generated classes. This was done as AutoMed is compatible with this version and any differences would reduce the stability and compatibility of the translator.

7.3 Structure

The parser is constructed as two separate Java files, a JavaCC and a main Java file for executing the translator and passing the IQL query back to the requestor. The JavaCC compiler creates, as part of the compilation process, multiple Java classes for parsing and error handling. The resulting files may be manually edited to improve the quality of the translation process.

Created
parseSql.jj
ParserMain.Java

7-1 Created Java Classes

Generated
ParseException.Java
parseSqlConstants.Java
parseSQLTokenManager.Java
SimpleCharStream.Java
Token.Java
TokenMgrError.Java

7-2 - Generated Java Classes

The main class, ParserMain.Java, contains the methods needed to receive an SQL statement from another program or command line user. This class, when run independently prompts the user for a complete SQL statement which is determined by checking for the input of a semicolon. It then invokes the parserSQL class which translates the SQL and returns the IQL to the requestor. The error handling classes ParseException.Java and TokenMgrError.Java are used to catch and report errors from the parsing process. The generic error reporting features of JavaCC are further improved to provide a less generic error message and aid the user in identifying where problems were detected in the SQL statement. This is achieved by caching the user input and comparing this with the location of any lexical or parser errors reported by the translator.

The parser was incrementally developed to allow for a smoother development process and for the developer to adjust to, and fully appreciate the JavaCC environment. The development involved the following stages:

- Identify tokens and keywords
- Create specification for basic SQL statement: $SELECT\ FROM$
- Add support for aggregation
- Add support for nested queries in $FROM$ clause
- Add support for set operators
- Add support for $WHERE$ clause
- Add support for nested queries in $WHERE$ clause
- Add support for $GROUP\ BY$ clause
- Implement parsing rules in the same order

7.4 Grammar

The first stage of the implementation was to define the grammar of the translator. The grammar is constructed as a JavaCC class file which is then compiled into a Java class file. The structure of the grammar consists of the options, class declaration, lexical specification and the syntax specification. The options section, defines any parser specific option including cases sensitivity and debugging options. The parser was set to ignore case as SQL has no cases sensitivity.

The class declaration section was used to declare any methods needed by the translator. The lexical analyser required no additional methods; however, this section was used for the translation process.

The lexical specification consists of both terminals and non-terminals defining the rules of SQL. The grammar was created from a base SQL grammar created by John Kristian and obtained from the JavaCC repository at https://Javacc.dev.Java.net/files/documents/17/38413/PlSql.jj. This was aimed at PL/SQL and was modified to remove unnecessary code and lexical rules. The grammar declares all keywords using the notation <K_SELECT> and all operators using the notation <S_QUOTED_STRING>. This reduces any ambiguities in writing the grammar and makes it easier to understand and develop.

The keywords were used to create the rules of the grammar using the standard JavaCC notation:

$$String\ SQLSimpleExpression(): \{String\ s1;\}$$

$$
\begin{array}{l}
\{ \\
\quad s1 \\
= SQLSimpleExpressions() \\
\\
\{\boldsymbol{return}\ s1;\} \\
\\
\}
\end{array}
$$

The grammar was kept simplified and modular, enabling the reuse of non-terminals. The grammar kept strict, with local lookahead values defined to ensure that the correct token was chosen on scanning. In most instances, the standard lookahead value of two was sufficient to resolve any localized problems. The ordering of grammar was not a factor and the components of SQL clauses were kept together making it easier to develop and read.

The design of the grammar incurred relatively few problems and any issues were easily debugged by enabling the debugging options in the file. After successfully specifying the grammar, the translation rules were applied.

7.5 Translation

The JavaCC grammar allows the embedding of translation rules into the lexical analysis stage. Each non-terminal was implemented as a Java method which allowed for the embedding of Java code to handle the translation rules. The translation process was done in several stages. Firstly, on being invoked and passed an SQL statement, a lexical analysis is performed to check for statement structure and validity against the specified grammar. The SELECT clause is used to derive the attributes and the source relations. Each item in the select clause is determined using the non-terminal:

```
LinkedList SelectItem():
{String s1, s2;
 Token t1;
LinkedList selectList = new LinkedList();
s1 = "";}
{

(
(
      //the select item is a fully qualified table
      s1 = RelObjectName()
      {
```

```
      selectList.add(s1);
      }

      "."

      s2 = DotObjectName()
      {
      selectList.add(s2);
      }
  )

  |
  //or a number
   t1=<S_NUMBER>

   {
      selectList.add(t1.image.toString());

   }
  )
{
      return selectList;}
}
//valid arithmetic operators.
String arithmeticOp():
{
Token t1;
}
{
(
      t1 = "+" | t1="*" | t1="/" | t1="-"
)
```

The select item can be a fully qualified table column or a number. Numbers are allowed for instances where aggregation may be applied to a column. This rule is strictly enforced and no other variables are allowed as in the select clause. The returned column is used to build a list of select items and any aggregation applied as shown below.

```
[s1=operator()]


  ["("]
       tempList=SelectItem()
            {
            selectList.add(s1 + "|" + tempList.getFirst().toString() +
"." + tempList.getLast().toString());
            s1 = null;
            if (tempList.size() > 1)
             {
                  fromList.add(tempList.getFirst().toString() + "|" +
tempList.getLast().toString());
             }
            }
       //some arithmetic is allowed
       [s1=arithmeticOp(){s2=selectList.getLast().toString();
selectList.removeLast(); selectList.add(s2 + s1);}
```

```
        //second item in the arithmetic equation
        tempList=SelectItem(){
             s2=selectList.getLast().toString();
selectList.removeLast();selectList.add(s2 +
tempList.getFirst().toString() + "." + tempList.getLast().toString());
            if (tempList.size() > 1)
             {
                  fromList.add(tempList.getFirst().toString() + "|" +
tempList.getLast().toString());
             }

              }]

      [")"]
      //multiple select items allowed.


returnList = createHead(selectList);
returnList.put("FROMLIST", new LinkedList(fromList));
returnList.put("SELECTLIST", new LinkedList(selectList));
return returnList;
}
```

The select item is used to create two lists; the *SELECTLIST* list is used for creating the head of the IQL query while the *FROMLIST* is used to create the body of the comprehension. Any aggregation functions (s1=operator (), s1=arithmeticOp ()) applied are also added to the *SELECTLIST* to be used in creating the IQL head. Operators are delimited using the "|" character which separates from the column names.

The translator then attempts to build the IQL head. This is created using a method in the parser class but outside the grammar specification. It takes as a single argument, the *SELECTLIST* and assembles the parts of the head as shown below:

```
  public static HashMap createHead(LinkedList selectList)
  {
      for (int i=0; i < selectList.size(); i++)
      {
            a = selectList.get(i).toString().indexOf ("|");

            b = selectList.get(i).toString().indexOf (".");

            tempList.add(selectList.get(i).toString().substring(b+1));

            s1 = selectList.get(i).toString().substring(a+1);

  if (agregList.size() > 1)
  {
  …***code omitted***…
}
  else if (agregList.size() < 1)
       {

       …***code omitted***…
```

```
else if ((agregList.size() < 2) && (agregList.size() > 0))
      {

         …***code omitted***…

          agregReturn.add(iqlHead1.toString());
    }
      }

    returnMap.put("IQLHEAD", new LinkedList(agregReturn));

    returnMap.put("IQLCOLUMNS", new LinkedList(tempList));

    return returnMap;
}
```

The method firstly creates two lists, one with all the attributes (selectList) and the second list (agregList) with all the attributes requiring an aggregation function. It then checks for the number of attributes with aggregation functions applied, and decides how the IQL head should be created. There are three distinct ways in which the head is created: where there is no aggregation it is of the format $\{a, b, c, \}\,|$, where there is a single aggregation it is of the format $sum[\{a, \{b, c\}|$. Where there are multiple aggregation functions, the IQL head is unable to accommodate this. The lower level of IQL requires that a comprehension is created for aggregate function. The comprehensions are then nested within a single comprehension and the aggregated columns returned

$sum[\{y, \{x, z\}\}\,|sum[\{y, \{x, z\}\}\,|\,sum[\{z, \{y, x\}\}\,|$

This method returns the IQL head along with the list of attributes. This list of attributes is used to resolve the group by clause of the query. The group by clause is created by applying a similar set of rules; however the *group* or *gc* functions are appended.

The translator then proceeds to build the comprehension using the from and where clauses. It uses the list of tables in the *SELECTLIST* along with a list of nested queries to resolve the comprehension. The translator assumes that all attributes in the select clause must be in the from clause and as such does not check for this.

```
                for (int i=0; i < bodyList.size(); i++)
        {

    a = bodyList.get(i).toString().indexOf ("|");
     //create list of relations and attributes
          String fromTable_a = bodyList.get(i).toString().substring(0,a);
          String fromTable_b = bodyList.get(i).toString().substring(a+1);

    boolean existsInMap = tableList.containsValue(fromTable_a);
          if (existsInMap == false)
          {
          tableList.put(i, new String (fromTable_a));
```

```
            }

***code omitted***

                  //iterate through the list of relations and build
generators
            while (iT.hasNext())
            {

            Object o1 = iT.next();


            String tbl = tableList.get(o1).toString().trim();

            outer:

            for (int j=0; j < columnList.length; j++)
            {

            if (columnList[j][0].equals(tbl))
                  {
            //if the current relations is an aliased nested query, build
the generator for this
                  if (nestedList.get(tbl)!=null)
                   {
                   colList = nestedList.get(tbl + "COLUMNS");

                        iqlBuffer.append("{");

                        for (int k=0; k < colList.size();  k++)

                        {
                        ***code omitted***
}
                  //otherwise build a standard generator
                  else
                  {

***code omitted***

                  }
                              }

            }
            }
***code omitted***
      return iqlBody;
  }
```

The method used to create the generators, accepts as its arguments, a HashMap of the nested queries (nestedList) and the *SELECTLIST*. In the code snippet above, it first uses the select list to derive the attributes and the relations to be used in the generators. The translator then attempts to iterate through the list of relations and their attributes, building generators for each. When building the generator, the translator uses the HashMap of nested queries to determine whether the relation

in the current scheme is an aliased nested query. If it is, then the associated query is substituted instead of the scheme. The generators, called iqlBody in the code, are returned as single String.

The *where* clause is then resolved, with little changes made. Syntactic differences are resolved such as the *in* function being converted to the IQL *member* function. The *group by* clause is then resolved. This modifies the select clause to add groupings and any group level aggregation required. The translation differences between IQL and SQL are discussed in greater detail in section 6 above. The code snippet below, shows the simpler transformations applied to the where clause.

```
	(s1=WhereItem())

	[(s2=NullWhere() | s2=numericComparisonOperator(){s2 =
Comp.get(s2.trim()).toString().toLowerCase();}  |
s2=stringComparisonOperator(){s2 =
Comp.get(s2.trim().toLowerCase()).toString();})

	(s3=WhereItem()
	{
System.out.println(s3);
//handle set membership in the where clause
		if (s2.toLowerCase().trim().equals("in"))
		{

			s4 = " member " + s3 + " " + s1;

		}


		else if (s2.toLowerCase().trim().equals("not in"))
		{

			s4 =  " not ( member " + s3 + " " + s1 + ")";

		}


//Handles null value comparisons in the where clause.
		else if (s3.toLowerCase().trim().equals("null"))
		{
			if (s2.toLowerCase().trim().equals("is"))
			{
				s4 =  s1 + " = Null ";
			}
			else if (s2.toLowerCase().trim().equals("is not"))
			{
				s4 =  s1 + " <> Null ";
			}

		}

		else
		{

//return the where clause
			s4 = s1 + s2 + s3;
```

```
            }

        })
        ]


{
        return s4;}
}
```

The where clause resolves differences in the structure of filters testing set membership as well as resolving the correct IQL comparison operator. The operators are translated using a Hash Map, which acts as a lookup table, substituting the correct operators.

Finally, the translator combines all three components into a single IQL query and returns this, which is then passed onto the AQP.

In order to successfully parse the statement, some symbols are used as delimiters and identifiers. The reserved word of the resulting translator is: "|" which is used as a list delimiter as well as the SQL and IQL reserved words list. The parser was implemented as discussed above and tested thoroughly with the results and comments detailed in chapter 8 below

7.6 Results

The translator component requires two main methods in order to integrate with AutoMed. The first method $getIQL()$ is used to translate and return the translated IQL query to the AQP. The second method $translateResult(ASG\ g)$ is used to translate the query results to an SQL compatible type. The method used to translate the results is currently being implemented. This is a relatively trivial part of the implementation process as results can still be viewed in IQL format.

7.7 Error Reporting

The translator implements two types of error reports. This achieved by using the parserMain class to call the translator as a runtime object. This allowed for the error messages to be customised and shown where possible.

```
} catch (ParseException e) {

              int line_index = inputQuery.lastIndexOf("\n");

                if (line_index> 0)
                {
```

```
//attempt to return a useful error message, if this fails return a
generic message as returned from the parser.
                    try {
                            System.out.println("Error After: " +
inputQuery.substring(0,(line_index + e.getCol_num())) + " *** ");
                        } catch (RuntimeException e1) {

                                System.out.println("Error After: " +
inputQuery.substring(0,(e.getCol_num())) + " *** ");

                        }
```

As shown in the code above, the translator maintains a copy of original input query (inputQuery) before parsing and uses that to return a more precise error message. For command line usage, outside of the AutoMed toolkit, the translator is then reinitialized using the code below and prompts for another input query.

```
//reinitialise the parser and prompt for another SQL statement.
      inputQuery = inString();
        parseSQL.ReInit(parseStringToIS(inputQuery));
```

7.8 Supported Query Types

The translator was created to support as a large a subset as possibly from the SQL ANSI 1992 standard. The development time and effort needed, meant it was kept confined to a more focused subset. The subset of queries supported includes:

The SELECT FROM WHERE GROUP BY structure.

The select clause requires fully qualified column names of the form table.name. It allows for aggregation over the columns returned as well as arithmetic operations involving one ore more columns. Aggregation and arithmetic functions supported are:

sum
count

avg
max
min
+
-
*
/

7-3 Supported Arithmetic and Aggregation Functions

The FROM clause supports the use of both tables and nested queries. Queries can be arbitrarily nested; however nested queries with aggregate columns would return longer columns names since they cannot be aliased. Therefore a nested query which performs some aggregation on two columns {a, b} would have the resulting column referenced in the outer query as {a_b}. This holds true for nested queries in the WHERE clause. Correlated queries are not supported by the translator in both the FROM and WHERE clauses. The translator supports the standard WHERE clause, including set membership and comparisons. However it does not support the SQL functions “EXISTS” or “BETWEEN” as well as their negated forms. The GROUP BY clause is supported by the translator, however aggregation is only allowed over a single column and the HAVING clause is not supported. The translator supports the distinct function across the entire result set, however does not allow a count (*) as is used in SQL without specifying a column. The list below shows the general types of queries supported:

Simple SELECT FROM
Simple SELECT FROM WHERE (Evaluating Numbers)
Simple SELECT FROM WHERE (Evaluating Strings)
Nested Queries in WHERE Clause.
Set Operators in the Outer Statement
Nested Queries in the FROM Clause
Select With Aggregation
Select With Group By
Nested Set Operators
Aggregation over Grouping

7-4 Supported Query Types

8 TESTING

Testing of the translator was done in stages, progressively testing the capabilities of the translator. IQL queries were created for the University databases supplied with the standard distribution of AutoMed. Queries were written in SQL to specifically test each implemented query type and its conversion to the IQL language. The testing phase was not done with the translator integrated into the AutoMed system but rather as a standalone translator, accepting inputs from the command line and returning the IQL equivalent.

The testing of the translator, does not involve the successful execution of the queries as this is outside the scope of the project. The testing phase checks the validity of the IQL equivalent comprehension and the resulting IQL queries cannot therefore be guaranteed to return actual results from the university databases. Owing to the size and schema of the database, the SQL statements used may not be optimized and in some cases, could be further simplified while remaining SQL compliant. However, this is also outside the scope of the project, and the statements though appearing to be badly constructed, are simply used to establish the translation accuracy. Some SQL queries are inaccurate by definition and are used to check the strictness and error trapping capabilities of the translator. This noted where applicable in the test results.

The accuracy of the IQL queries in each instance was verified by attempting to create an Abstract Syntax Graph (ASG) representation of the query. This was achieved using the following code:

```
public static void main (String[] args)
{
System.out.println("Checking Translation ");

String[] queries = {
"SELECT dept.dname FROM dept;",
"SELECT female.id FROM female;",
"SELECT male.id FROM male;"

};

ASG g;
for(int i = 0; i < queries.length; i++) {
 try{
  g = new ASG(queries[i]);
 } catch(Exception e) {
  System.out.println("Syntax error in query: " + queries[i]);
  e.printStackTrace();
 }
}

System.out.println("All Done! ");

}
```

Successfully creating an ASG indicates that the IQL query is syntactically correct; however it does not test its ability to be executed across the schema.

8.1 Testing Results

8.1.1 Simple SELECT FROM

Single Attribute Returned:

SQL A

$SELECT\ dept.dname$
$FROM\ dept;$

IQL A

$[\{\ dept\_dname\ \} |\quad \{deptPK, dept\_dname\} \leftarrow \langle\langle dept, dname\rangle\rangle\ ]$

Multiple Attributes Returned:

SQL B

$SELECT\ degree.dcode, degree.dname, degree.title$
$FROM\ degree;$

IQL B

$$[\{\ degree\_dcode,\ degree\_dname,\ degree\_title\ \} |\ \{degreePK, degree\_dcode\} \leftarrow \langle\langle degree, dcode\rangle\rangle;\ \{degreePK, degree\_dname\} \leftarrow \langle\langle degree, dname\rangle\rangle;\ \{degreePK, degree\_title\} \leftarrow \langle\langle degree, title\rangle\rangle\ ]$$

Both queries were translated as expected. The *PK* annotation is used to generically refer to the key of the schema being used for the query. The attributes returned for each tuple is translated using the fully qualified notation to reduce any potential ambiguities as discussed in section 7.

8.1.2 Simple SELECT FROM WHERE (Evaluating Numbers)

This checks the ability to translate a SELECT FROM WHERE statement where a number is being evaluated in the WHERE clause. Translated correctly, the where clause is appended to the comprehension as the filter separated by a semi-colon.

Single Attribute Evaluated:

SQL A

$SELECT\ person.id, person.dname, person.name$
$FROM\ person$
$WHERE\ person.id\ =\ 4;$

IQL A

$[\{\ person\_id,$
$person\_dname,$
$person\_name\ \}\ |\ \{personPK, person\_id\}\ \leftarrow\ \langle\langle person, id\rangle\rangle;$
$\{personPK, person\_dname\}\ \leftarrow\ \langle\langle person, dname\rangle\rangle;$
$\{personPK, person\_name\}\ \leftarrow\ \langle\langle person, name\rangle\rangle;$
$person\_id\ =\ 4\ ]$

Multiple Attributes Evaluated:

SQL B

$SELECT\ person.id, person.dname, person.name$
$FROM\ person$
$WHERE\ person.id\ =\ 4\ or\ person.id\ =\ 6;$

IQL B

$[\{\ person\_id,$
$person\_dname,$
$person\_name\ \}\ |\ \{personPK, person\_id\}\ \leftarrow\ \langle\langle person, id\rangle\rangle;$
$\{personPK, person\_dname\} \leftarrow\ \langle\langle person, dname\rangle\rangle;$
$\{personPK, person\_name\}\ \leftarrow\ \langle\langle person, name\rangle\rangle;$
$person\_id\ =\ 4\ or\ person\_id\ =\ 6\ ]$

Queries were translated as expected. Both attributes were included in the filter.

8.1.3 Simple SELECT FROM WHERE (Evaluating Strings)

This checks the ability to translate a SELECT FROM WHERE statement where a string is evaluated in the WHERE clause. Translated correctly, the where clause is appended to the comprehension as the filter separated by a semi-colon.

Single Attribute Evaluated:

SQL A

$SELECT\ dept.dname$
$FROM\ dept$
$WHERE\ dept.dname\ <>\ 'Computing - IC';$

IQL A

$[\{\ dept\_dname\ \}\ |\quad \{deptPK, dept\_dname\} \leftarrow \langle\langle dept, dname\rangle\rangle;$
$dept\_dname\ <>\ 'Computing - IC'\ ]$

Multiple Attributes Evaluated:

SQL B

$SELECT\ dept.dname$
$FROM\ dept$
$WHERE\ dept.dname\ like\ 'Math\%'$
$AND\ dept.dname\ <>\ 'Computing - IC'$
$AND\ dept.dname\ <>\ 'English';$

IQL B

$[\{\ dept\_dname\ \}\ |\quad \{deptPK, dept\_dname\} \leftarrow \langle\langle dept, dname\rangle\rangle;$
$dept\_dname\ like\ 'Math\%'$
$and\ \ dept\_dname\ <>\ '\ Computing - IC'$
$and\ dept\_dname\ <>\ 'English';]$

The SQL queries were translated correctly. Like SQL, strings are enclosed in single quotes and the pattern matching allowed using the "*". The LIKE operator remains unchanged and translated directly across to IQL.

8.1.4 Nested Queries in WHERE Clause.

This tests the ability to translate queries where there is a nested of derived query in the WHERE clause. Nested queries in the WHERE clause typically checks for set membership and should allow arbitrary nesting.

Single Nested Query

SQL A

$SELECT\ degree.dcode, degree.title, degree.dname$
$FROM\ degree$
$WHERE\ degree.dname\ NOT\ IN$
(
$SELECT\ dept.dname$
$FROM\ dept$
$WHERE\ dept.dname\ <>\ 'Computing - IC'$
) ;

IQL A

[{ $degree\_dcode$,
$degree\_title$,
$degree\_dname$} | $\{degreePK, degree\_dcode\} \leftarrow \langle\langle degree, dcode\rangle\rangle$;
$\{degreePK, degree\_title\} \leftarrow \langle\langle degree, title\rangle\rangle$;
$\{degreePK, degree\_dname\} \leftarrow \langle\langle degree, dname\rangle\rangle$;
not ($member$
[{ $dept\_dname$ } | $\{deptPK, dept.dname\} \leftarrow \langle\langle dept, dname\rangle\rangle$;
$dept.dname\ <>\ 'Computing - IC'$]
$degree\_dname$)]

Multiple Nested Queries

SQL B

$SELECT\ degree.dcode, degree.title, degree.dname$
$FROM\ degree$
$WHERE\ degree.dname\ NOT\ IN$
(
$SELECT\ dept.dname$
$FROM\ dept$

$WHERE\ dept.dname\ <>\ 'Computing - IC'$

)

AND

$degree.dname\ IN$

(

$SELECT\ dept.dname$

$FROM\ dept$

$WHERE\ dept.dname\ =\ 'Math - IC'$

);

IQL B

$[\{\ degree\_dcode,$

$degree\_title,$

$degree\_dname\}\ |\quad \{degreePK, degree\_dcode\} \leftarrow \langle\langle degree, dcode\rangle\rangle;$

$\{degreePK, degree\_title\} \leftarrow \langle\langle degree, title\rangle\rangle;$

$\{degreePK, degree\_dname\} \leftarrow \langle\langle degree, dname\rangle\rangle;$

$not\ (\ member$

$[\{\ dept\_dname\ \}\ |\quad \{deptPK, dept\_dname\} \leftarrow \langle\langle dept, dname\rangle\rangle;$

$dept\_dname\ <>\ 'Computing - IC'\ ]$

$degree\_dname)\ ]\ and$

$member$

$[\{\ dept\_dname\ \}\ |\quad \{deptPK, dept\_dname\} \leftarrow \langle\langle dept, dname\rangle\rangle;$

$dept\_dname\ =\ 'Math - IC'\ ]\ degree\_dname\ ]$

The nested queries in the where clauses are translated correctly. The *IN* function of SQL is correctly translated to the *member* function of IQL. The logical operator is correctly translated and the functions are translated into lowercase as required by IQL. These queries are of the correct syntax and would be correctly executed against the schema. This query could be further optimized to use a single query joining both tables; however for the purpose of the translation testing it is accurate.

8.1.5 Set Operators in the Outer Statement

This tests for the ability to translated set operations in the outermost query. The set operation translation is relatively straightforward.

Single Set Operator

SQL A

```
SELECT person.id, person.name, person.dname
FROM person
UNION ALL
(
SELECT staff.id, staff.name, staff.dname
FROM staff
);
```

IQL A

```
[ { person_id,
  person_name,
person_dname } |   {personPK, person_id} ← ‹‹person, id››;
                   {personPK, person_name} ← ‹‹person, name››;
                   {personPK, person_dname} ← ‹‹person, dname›› ]
++
([ { staff_id,
staff_name,
staff_dname } |    {staffPK, staff_id} ← ‹‹staff, id››;
                   {staffPK, staff_name} ← ‹‹staff, name››;
                   {staffPK, staff_dname} ← ‹‹staff, dname›› ])
```

Multiple Set Operators

SQL B

```
SELECT person.id, person.name, person.dname
FROM person
UNION ALL
(
SELECT staff.id, staff.name, staff.dname
FROM staff
INTERSECT
(
```

$SELECT\ person.id, person.name, person.dname$

$FROM\ person$

)

);

IQL A

$[\{ person\_id,$
$person\_name,$
$person\_dname\ \}\ |\ \{personPK, person\_id\} \leftarrow \langle\langle person, id\rangle\rangle;$
$\{personPK, person\_name\} \leftarrow \langle\langle person, name\rangle\rangle;$
$\{personPK, person\_dname\} \leftarrow \langle\langle person, dname\rangle\rangle\ ]$

++

$([\{ staff\_id,$
$staff\_name,$
$staff\_dname\ \}\ |\ \{staffPK, staff\_id\} \leftarrow \langle\langle staff, id\rangle\rangle;$
$\{staffPK, staff\_name\} \leftarrow \langle\langle staff, name\rangle\rangle;$
$\{staffPK, staff\_dname\} \leftarrow \langle\langle staff, dname\rangle\rangle\ ]$

intersect

$([\{ person\_id,$
$person\_name,$
$person\_dname\ \}\ |\ \{personPK, person\_id\} \leftarrow \langle\langle person, id\rangle\rangle;$
$\{personPK, person\_name\} \leftarrow \langle\langle person, name\rangle\rangle;$
$\{personPK, person\_dname\} \leftarrow \langle\langle person, dname\rangle\rangle\ ]))$

Set operations were translated correctly as expected. Operators were converted to the correct lower case and the parenthesis maintained.

8.1.6 Nested Queries in the FROM Clause

This checks for the ability of the translator to correctly translate nested queries in the *FROM* clause of the SQL statement. As discussed in section 5, IQL fully supports nested queries. Where queries are nested, all the attributes returned by the inner query must be handled in the outer query (see section 6). The translation resolves this, allowing the user to continue specifying only the columns needed.

Single Nested Query

SQL A

```
SELECT people.name, people.dname
FROM
(
SELECT person.id, person.name, person.dname
FROM person
)AS people
WHERE people.dname LIKE 'Computing%';
```

IQL A

```
[ { people_name,
 people_dname } |    {people_id, people_name, people_dname} ←
                     [ { person_id,
                     person_name,
                      person_dname } |    {personPK, person.id} ← ‹‹person, id››;
                                          {personPK, person.name} ← ‹‹person, name››;
                                          {personPK, person.dname} ← ‹person, dname›› ];
                     people_dname like 'Computing%' ]
```

Multiple Nested Queries

SQL B

```
SELECT people.name, people.dname, dept.dname
FROM
(
SELECT person.id, person.name, person.dname
FROM person
)AS people,
(
SELECT dept.dname
FROM dept
WHERE dept.dname = 'Computing'
)AS dept;
```

IQL B

$$
\begin{array}{lll}
[\{\, people\_name, & & \\
\quad people\_dname, & & \\
\quad dept\_dname \,\} \mid & \{people\_id, people\_name, people\_dname\} \leftarrow & \\
& [\{\, person\_id, & \\
& \quad person\_name, & \\
& \quad person\_dname \,\} \mid & \{personPK, person\_id\} \leftarrow \langle\langle person, id \rangle\rangle; \\
& & \{personPK, person\_name\} \leftarrow \langle\langle person, name \rangle\rangle; \\
& & \{personPK, person\_dname\} \leftarrow \langle\langle person, dname \rangle\rangle\,]; \\
& \{dept\_dname\} <- & \\
& [\{\, dept\_dname \,\} \mid & \{deptPK, dept\_dname\} \leftarrow \langle\langle dept, dname \rangle\rangle; \\
& & dept\_dname = 'Computing'\,]\,]
\end{array}
$$

The nested queries where translated correctly and as expected. The nested tables were fully resolved and all the columns from the nested query referenced in the outer queries.

8.1.7 Select With Aggregation

This test checks for aggregation support in the translation process from SQL to IQL. Aggregation over a single attribute is relatively straightforward in the translation process; however, IQL's lower level implementation requires that multiple comprehensions are generated for each aggregation.

Single Column Aggregation

SQL A

$$
\begin{array}{l}
SELECT\ MAX\ (degree.id) \\
FROM\ degree \\
WHERE\ degree.id < 10;
\end{array}
$$

IQL A

$$
\begin{array}{lll}
max\,[\{\, degree\_id \,\} \mid & \{degreePK, degree\_id\} \leftarrow \langle\langle degree, id \rangle\rangle; \\
\quad degree\_id < 10\,]
\end{array}
$$

Multiple Column Aggregation

SQL B

$$
\begin{array}{l}
SELECT\ MAX(staff.name), MIN(staff.id), staff.dname \\
FROM\ staff \\
WHERE\ staff.name <> 'Jamie';
\end{array}
$$

IQL B

$[\{ staff\_name,$
$staff\_id,$
$staff\_dname \} |$ $\{ staff\_name \} \leftarrow$
$max [\{ staff\_name \} |$ $\{staffPK, staff\_name\} \leftarrow \langle\langle staff, name\rangle\rangle;$
$\{staffPK, staff\_id\} \leftarrow \langle\langle staff, id\rangle\rangle;$
$\{staffPK, staff\_dname\} \leftarrow \langle\langle staff, dname\rangle\rangle;$
$staff\_name <> 'Jamie'];$
$\{ staff\_id \} \leftarrow$
$min [\{ staff\_id \} |$ $\{staffPK, staff\_name\} \leftarrow \langle\langle staff, name\rangle\rangle;$
$\{staffPK, staff\_id\} \leftarrow \langle\langle staff, id\rangle\rangle;$
$\{staffPK, staff\_dname\} \leftarrow \langle\langle staff, dname\rangle\rangle;$
$staff\_name <> 'Jamie']]$

Aggregation over single and multiple columns translated as expected.

8.1.8 Select With Group By

This tests the ability of the translator to handle SQL statements with a *GROUP BY* clause.

Grouping by Single Attribute

SQL A

$SELECT\ degree.dcode, degree.dname, degree.title$
$FROM\ degree$
$GROUP\ BY\ degree.dcode;$

IQL A

$group$
$[\{degree\_dcode,$
$\{degree\_dname,$
$degree\_title \}\} |$ $\{degreePK, degree\_dcode\} \leftarrow \langle\langle degree, dcode\rangle\rangle;$
$\{degreePK, degree\_dname\} \leftarrow \langle\langle degree, dname\rangle\rangle;$
$\{degreePK, degree\_title\} \leftarrow \langle\langle degree, title\rangle\rangle]$

Grouping by Multiple Attributes

SQL B

$SELECT\ degree.dcode, degree.dname, degree.title$
$FROM\ degree$
$GROUP\ BY\ degree.dcode, degree.dname;$

IQL B

$group$
$[\{\{degree\_dcode,$
$degree\_dname\},$
$degree\_title\}\ |\quad \{degreePK, degree\_dcode\} \leftarrow \langle\langle degree, dcode\rangle\rangle;$
$\{degreePK, degree\_dname\} \leftarrow \langle\langle degree, dname\rangle\rangle;$
$\{degreePK, degree\_title\} \leftarrow \langle\langle degree, title\rangle\rangle\ ]$

GROUP BY clause translated correctly and as expected.

8.1.9 Nested Set Operators

This tests the ability of the translator is correctly translate SQL statements with set operations nested in the from clause. The set operators should be correctly substituted.

SQL A

$SELECT\ people.name, people.dname$
$FROM$
$($
$SELECT\ person.id, person.name, person.dname$
$FROM\ person$
$UNION\ ALL$
$(SELECT\ staff.id, staff.name, staff.dname$
$FROM\ staff)$
$)AS\ people$
$WHERE\ people.dname\ LIKE\ \ 'Computing\%';$

IQL A

$[\{\ people\_name,$
$people\_dname\ \}\ |\quad \{people\_id, people\_dname, people\_name\} \leftarrow$
$[\{\ person\_id,$
$person\_name,$

```
 person_dname } |     {personPK, person_id} ← ‹‹person, id››;
                      {personPK, person_name} ← ‹‹person, name››;
                      {personPK, person_dname} ← ‹‹person, dname›› ]
++
([ { staff_id,
staff_name,
staff_dname } |       {staffPK, staff_id} ← ‹‹staff, id››;
                      {staffPK, staff_name} ← ‹‹staff, name››;
                      staffPK, staff_dname} ← ‹‹staff, dname›› ]);
people_dname like    'Computing%' ]
```

SQL B

```
SELECT people.name, people.dname
FROM
(
SELECT person.id, person.name, person.dname
FROM person
MINUS
(SELECT staff.id, staff.name, staff.dname
FROM staff)
)AS people
WHERE people.dname LIKE 'Computing*';
```

IQL B

IQL A

```
 [ { people_name,
 people_dname } |    {people_id, people_dname, people_name} ←
                     [ { person_id,
                      person_name,
                      person_dname } |    {personPK, person_id} ← ‹‹person, id››;
                                          {personPK, person_name} ← ‹‹person, name››;
                                          {personPK, person_dname} ← ‹‹person, dname›› ]
                     --
                     ([ { staff_id,
                     staff_name,
                     staff_dname } |      {staffPK, staff_id} ← ‹‹staff, id››;
```

$$\{staffPK, staff\_name\} \leftarrow \langle\langle staff, name\rangle\rangle;$$
$$staffPK, staff\_dname\} \leftarrow \langle\langle staff, dname\rangle\rangle \,]);$$
$$people\_dname\ like \quad 'Computing\%' \,]$$

The results were as expected and the queries were correctly translated.

8.1.10 Aggregation over Grouping

This tests the ability of the translator is correctly translate SQL statements with aggregation over grouped attributes.

SQL A

$$SELECT\ MAX(degree.dcode), degree.dname, degree.title$$
$$FROM\ degree$$
$$GROUP\ BY\ degree.dname;$$

IQL A

$$gc\ \max[\{\{degree\_dname\},$$
$$degree.dcode\}\ |\quad degreePK, degree\_dcode\} < - << degree, dcode >>;$$
$$\{degreePK, degree\_dname\} < - << degree, dname >>;$$
$$\{degreePK, degree\_title\} < - << degree, title >>\,]$$

SQL B

$$SELECT\ degree.dcode, degree.dname, MAX(degree.title)$$
$$FROM\ degree$$
$$GROUP\ BY\ degree.dcode, degree.dname;$$

IQL B

$$gc\ \max$$
$$[\{\{degree\_dcode, degree\_dname\}$$
$$, degree.title\}\ |\quad \{degreePK, degree\_dcode\} < - << degree, dcode >>;$$
$$\{degreePK, degree\_dname\} < - << degree, dname >>;$$
$$\{degreePK, degree\_title\} < - << degree, title >>\,]$$

The SQL statements were translated correctly and as expected.

8.1.11 Statement Validity Checking

This section tests the statement validity checking of the lexical analyser and parser. All the statements used are invalid SQL statements and should generate and error message from the translator. The translator firstly attempts to report the exact location of the error with a more friendly error message and asterisks at the point of the error. In some instances, the location of the error cannot be correctly determined and the entire statement is returned instead. This is a bug in the error reporting feature of the translator and does not affect its overall accuracy.

Test A

$SELECT\ degree.dcode, degree, degree.title$
$FROM\ degree;$

Result A

```
Error After:
SELECT degree.dcode,degree ***
```

As expected, an error was returned for column 28. The translator requires fully qualified column references (see section 7).

Test B

$SELECT\ degree.dcode, degree.dname, degree.title;$

Result B

```
Error After:
SELECT degree.dcode, degree.dname, degree.title*
```

Error correctly reported. The SQL statement is incomplete as a FROM clause was not included. The translator is expecting this after the list of tables.

Test C

$SELECT\ people.name, people.dname, dept.dname$
$FROM$
(
$SELECT\ person.id, person.name, person.dname$

$FROM\ person$

),

(

$SELECT\ dept.dname$

$FROM\ dept$

$WHERE\ dept.dname\ =\ 'Computing'$

$)AS\ dept$

;

Result C

```
Error After:
SELECT people.name, people.dname, dept.dname
FROM
(
SELECT person.id, person.name, person.dname
FROM person
),
(
SELECT dept.dname
FROM dept
WHERE dept.dname = 'Computing'
)AS dept
; ***
```

Error correctly detected in the SQL statement. The derived query was not correctly aliased. However the, error reporting tool was unable to correctly report the location of the error. In such a case, the entire input statement is returned.

Test D

$SELECT\ degree.dcode, degree.dname, degree.title$

$FROM\ degree$

$GROUP\ BY\ degree.dcode\ degree.dname;$

Result D

```
Error After:
SELECT degree.dcode, degree.dname, degree.title
FROM degree
GROUP BY degree.dcode  ***
```

The missing comma separator in the list of group by tables was correctly reported. The translator is expecting either a separator or the terminal semi-colon character.

Test E

$SELECT\ MAX(staff.name), staff.dname$
$FROM\ staff$
$WHERE\ staff.sex\ <>\ 'F;$

Result E

```
Error After:
SELECT MAX(staff.name), staff.dname
FROM staff
WHERE staff.sex <>  ***
```

The translator has correctly reported the incomplete SQL statement. There is a comparison operator with no valid comparison field such as a number, string or column reference. The location of the error has been correctly reported.

Test F

$SELECT\ people.name, people.dname$
$FROM$
(
$SELECT\ person.id, person.name, person.dname$
$FROM\ person$
$INTESECT$
$(SELECT\ staff.id, staff.name, staff.dname$
$FROM\ staff)$
$)AS\ people$
$WHERE\ people.dname\ LIKE\ 'Computing*';$

Result F

```
Error After:
SELECT people.name, people.dname
FROM
(
SELECT person.id, person.name, person.dname
FROM person
INTESECT
(SELECT staff.id, staff.name, staff.dname
```

```
FROM staff)
)AS people
 ***
```

In this test, the keyword INTERSECT was incorrectly spelt. The translator correctly reported an error in the SQL statement, however, was unable to report the error location correctly. The complete input statement was returned included in the error report.

The testing was successful and produced the required results. The extended test results are contained in the appendix and includes the results listed above. Any accuracy in the execution of the queries will always be undermined if the user fails to adhere to the schema. The translator, having no means of confirming the validity of the query against the underlying schema, simply serves to produce an IQL equivalent of the user input. Further improvements on the translator could be used to resolve this, however, as it stands, the translator component meets the intended aims and objectives.

9 CONCLUSION

The aim of the project was to create an SQL to IQL translator for the AutoMed toolkit. The translator should be able to accept SQL queries as detailed in the implementation section, and convert these queries into the IQL equivalent for evaluation across the chosen schema. The background research carried out for the project was relatively detailed and allowed for a choice of feasible approaches.

Building on the chosen approach, the implementation process was relatively smooth. The construction of the grammar required a good grasp of EBNF and JavaCC development procedures; however these were easy to grasp and use. The grammar was developed using a publicly available SQL grammar which required extensive changes to the structure and types of non-terminals. However, it had clearly defined the terminal symbols and provided a good base for the specification of the non-terminals. JavaCC provided useful debugging options to continually check the accuracy of the translator. The grammar was kept as concise and accurate as possible to create a class with the smallest memory footprint while fulfilling requirements.

The implementation of the translation rules was met, with some problems with regards to the choice of data structures. The aim was to utilize data structures that would remain highly accurate, while requiring a relatively small amount of memory. This could reduce any potential errors the parser may encounter with memory leaks when implemented in a server environment with multiple concurrent executions. Strings, lists, maps and arrays were declared with minimal allocated memory and wherever possible were destroyed when not needed. This was also in keeping with good programming practices. The Eclipse JavaCC plug-in, during development and testing, was at times unreliable and failed to maintain the operating system environment by terminating instances of the Java compiler. This often resulted in memory issues, with the system failing after any amount of extended development. To resolve this issue a command line script was used to periodically terminate any orphaned compiler instances.

9.1 Critical Review and Analysis

9.1.1 Background Research

The background research was thorough relative to the aims of the project and created a well defined foundation for the selection of a suitable approach. The limited information available on JavaCC

negatively affected the ability to examine all the possible approaches to creating and optimising the grammar. However, from the information gleaned during the research, JavaCC surpassed the alternatives which were limited in compatibility as discussed in section 3 above.

9.1.2 Approach

While using only Java, rather than the JavaCC tool would have required more development time, the need to learn the JavaCC language would have been negated, allowing for faster development. However, JavaCC, while requiring the learning of a relatively new language, which was slow and time consuming, was a more compatible and easier to use development tool. It required fewer lines of code and produced a leaner, more robust program. The resulting parser requires no additional code or classes to be executed. The parser can be independently executed outside of the AutoMed toolkit and allows users to enter queries on the command line and obtain the retuned translation. This is a potentially useful feature as it can allow for practising and learning of IQL outside of the AutoMed environment such as in a standalone translator. It could be incorporated with other translators to create translators several languages including SQL →IQL→OQL or SQL→IQL→ XQUERY.

9.1.3 Input

The translator accepts SQL queries either as a single line queries or as a multi line query. This allows users to format the SQL queries similarly to most DBMSs available. Queries can be formatted in a more visually appealing way so that errors can be more readily detected and prevented. The queries can span as many lines as needed and complete query is read until a semi-colon is detected, again mirroring the user experience with industry standard SQL databases. Submitting queries using another program or file will also result in the number of lines being ignored and standard SQL formatting being accepted. The query input, however does not allow a user to backspace or undo a previous line entered. As a result of this, if an error is detected in a previous line the user will have to execute the query and then try again. A more interactive approach to this part of the process could be implemented.

9.1.4 Translation

The IQL grammar specification and translation rules were written together in a single file, this allowed for fast development and writing of the specification. However it created very long and complex code which was prone to bugs and errors; requiring more detailed and thorough testing. An

approach separating the lexical analysis and translation phases could have been adopted to remove this issue. Adequate commenting of the grammar and source files reduced the impact of the complex class files and should allow for easy maintenance and extension of the translator.

The translator can translate statements as described in section 7 above. SQL translations are currently restricted to a subset of the SQL 92 standard. While this is sufficient for most common queries, the translator would be unable to support more complex queries as would be needed across data warehousing environments which often includes the now standard ROLLUP function. The grammar however can be readily modified to accept any constructs of the higher level language.

One limitation of the translator is its ability to fetch and resolve the underlying schema. This could enable more complex queries as well as the validation of queries before being submitted to the AQP. This could serve to reduce query errors including the available tables, columns and data type and create more accurate queries over the global schema. The translator could cache the schema and automatically assign columns to tables. This would remove the requirement for fully qualified columns in the SELECT statement, as discussed in 6 above, allowing for more compact and efficient queries. This caching feature would however, reduce the overall speed of the translator as each translation would first require a query to the database and resolving the schema to SQL standard. As implemented, it relies on the users to submit correct queries based on the schema. This however, is not critical to the accuracy and functionality of the translator since the AQP checks for query validity against the schema.

The translator inherently enforces some level of strictness with respect to the grammar. This is achieved in some areas of the code where the risk of localized ambiguities is higher than normal or where the translator has difficulty distinguishing the tokens. The lookahead value is modified so that the grammar examines more or fewer tokens before making a decision. The grammar could be further modified to improve strictness of the lexical analyser and the translator for trivial areas such as agreement between comparison operators and the type of data being compared. This could be used in the where clause to ensure that string operators such as “like” would be followed by only string values.

The grammar was written using a modular approach, resulting in longer but more structured parser. As an advantage, this allows the grammar to be more readily extended, reusing defined methods, terminals and non-terminals.

The parser is implemented in two primary classes and other supporting classes. The main class creates and instance of the parser at runtime and interfaces with it to execute the translation

process. In doing this, all code non-parsing code is kept separate and in a smaller class file. As an advantage, this encapsulates the parser, and allows users to easily reuse the parser class without needed to make and modifications.

9.1.5 Results

By default, the parser only returns the resulting IQL query to the AQP for evaluation or to the user or viewing. As a component in AutoMed, the results would either be a result set or some error message thrown by the translator. The parser however, can be modified to enable a more verbose view of the translation process. This can be used for debugging or resolving and translation errors. Currently, this option has to be enabled by editing the source code and recompiling the classes. The potential for using command switches for toggling this option is discussed in section 9.2.

The translator meets its aims and objectives. It can successfully translate SQL statements from the subset described in the implementation. While there are sometimes, noticeably time differences between submitting a query and receiving its IQL equivalent, the translated queries are accurate and from the testing can be successfully queried against the underlying schema.

9.1.6 Error Reporting

The parser uses friendly error messages to encapsulate the more advance messages output. Instead of displaying detailed parsing reports on the consumption of tokens or the list of tokens expected, the user is presented with a simple error message indicating the likely position of the error detected within the SQL statement. The customised error messages could be modified to notify the user on the token that the parser was expecting and suggest a correction.

9.2 Future Work

The parser can be further developed to extend its support for the SQL language. Since IQL in AutoMed is primarily used as a database query tool, shot-term work to the translator could seek to implement rules to fully support the SELECT subset of the SQL DML. Key areas of this implementation would be the full support for nested queries with the same attributes appearing in both nested and outer query. Other features of SQL could be the support of the EXIST / NOT EXIST and the BETWEEN functions in the where clause. These features would improve the querying

capabilities of the SQL supported AutoMed implementation. Further changes in the short term could include support for the HAVING clause which could simply be implemented as a filter on a nested group-by comprehension. Currently, users are required to explicitly nest a group-by comprehension to achieve this.

In the medium term, the translator could be extended to add a pre-processing component. This would involve pre-fetching the underlying AutoMed schema and validating queries before translating and passing them to the AQP. This would remove the need to support only fully qualified attributes since the translator would be able to resolve the attributes to their tables and build the corresponding query based on this. The pre-processor could also be used to validate data types and attribute lengths, and reduce any errors being passed onto the AQP. The error handling and reporting classes could be further customised to provide different levels of error reporting based on user preferences. Since the translator is implemented with Java code embedded in the grammar, the rules for each non-terminal can be modified to throw specific exceptions detailing the terminals expected at the point of error. The translator could also be extended to accept command line switches to select error levels and any other allowed user options.

In the long term, the translator could be extended to support the INSERT, DELETE and UPDATE subsets of the DML. Supporting these features of SQL would require that the short and medium term proposals are implemented first since features such as pre-fetching the schemas would be required in order to reliably support function such as update and insert. This extension of the parser could be more reliably and efficiently implemented as separate classes to support each major feature of the SQL DML. This would maintain the initial aim of having source code which is easily modified and maintained. The parser could also be extended to support SQL beyond the ANSI 92 standard. This would allow for the translator to be more efficiently used in data warehousing as it could support the newer features of standard SQL such as window and group by rollup queries.

10 REFERENCES

1. J**asper, Edgar; Zamboulis, Lucas; Mittal, Sandeep; Fan, Hao; Poulovassili, Alexandra.** *Processing IQL Queries and Migrating Data in the AutoMed toolkit.* 2007. pp. 3 - 28.

2. **Poulovassilis, Alex and Zamboulis, Lucas.** *A Tutorial on the IQL Query Language.* 2007. pp. 1 - 12.

3. **Jiang, Tao; Li, Ming; Regan, Kenneth W; Bala, Ravikumar** *Formal Grammars and Languages.* pp. 6 - 10.

4. **Garshol, Lars M.** BNF and EBNF: What are they and how do they work? *Lars Marius Garshol.* [Online] 21 07 2003. [Cited: 23 07 2007.] http://www.garshol.priv.no/download/text/bnf.html.

5. **Story, Clifford.** Lexical Analysis. *www.mactech.com.* [Online] 29 July 2007. [Cited: 29 July 2007.] http://www.mactech.com/articles/mactech/Vol.06/06.04/LexicalAnalysis/index.html.

6. **Aho, Alfred V, Sethi, Ravi and Ullman, Jeffrey D.** *Compilers: Principles, Techniques and Tools.* s.l. : Addison-Wesley Publishing Company, 1986.

7. **Hernandez, Thomas and Subbarao, Kambhampati.** *Integration of Biological Sources: Current Systems and Challenges Ahead.* s.l. : Sigmod Record, 2004. pp. 1 - 5. Vol. 33.

8. **Calvanese, Diego; Guiseppe, De Giacomo; Lenzerini, Maurizio; Nardi, Daniele; Rosati, Riccardo.** *Data Integration in Data Warehousing.* International Journal of Cooperative Information Systems, 2001. Vol. 10.

9. **van der Wielen, Marc.** *Improving the quality of data integration systems using the Both-as-View approach. Emphasizing data lineage in integration systems.* 2005. pp. 16 - 30.

10. **Friedman, Marc, Levy, Alon and Millstein, Todd.** Navigational Plans for Data Integration. s.l. : AAAI/IAAI, 1999, pp. 1 - 2 .

11. **Boyd, Michael; Lazanitis, Charalambos; Kittivoravitkul, Sasivimol; McBrien, Peter; Rizopoulos, Nikos.** *An Overview of The AutoMed Repository.* London : Dept. of Computing, Imperial College, 2004.

12. **Jasper, Edgar; Tong, Nerissa; McBrien, Peter; Poulovassilis, Alexandra.** *Generating and Optimising Views from Both as View Data Integration Rules.* 2004. pp. 1 - 18.

13. **Silberschatz, Abraham, Korth, Henry and Sudarshan, S.** *Database Systems and Concepts.* s.l. : McGraw-Hill Book Co., 1997.

14. **Zamboulis, Lucas and Poulovassilis, Alex.** *A Tutorial on the IQL Query Language.* 2007. pp. 1 - 10.