SQL Front-End for the JRelix Relational-Programming System

Ibrahima KHAYA

School of Computer Science
McGill University, Montreal, Quebec, Canada

March 2008

A thesis submitted to McGill University in partial fulfilment of the
requirements of the degree of
Master of Science

T. H. Merrett, Advisor

Copyright © Ibrahima KHAYA 2008

Abstract

This thesis discusses the design and implementation of an SQL front end for a relational database programming system JRelix. The purpose of this thesis is to lay a strong basis for the full SQL support by JRelix.

The SQL language is the de facto standard in Relational Databases which motivates its integration into JRelix. Through the SQLSTMT and SQLEXP commands the user can use SQL to perform operations on a JRelix database. These operations are translated into Aldat (the native programming language of JRelix) and can interactively perform updates, deletions and selections on the distinct relations of the database. The SQLSTMT command allows us to execute full SQL commands on the system whereas the SQLEXP command returns a relational expression to the system that can interact with other elements of the commands written in Aldat in the same statement.

Résumé

La présente thèse traite de la conception et de la mise en œuvre d'une interface SQL pour le système de programmation de bases de données relationnelles JRelix. Cette thèse a pour but de poser les fondements d'une couverture exhaustive de SQL par JRelix.

Le Language SQL est le standard en bases de données relationnelles ce qui est à l'origine de son intégration dans JRelix. A l'aide des commandes SQLSTMT et SQLEXP l'utilisateur peut effectuer des requêtes sur la base de données JRelix en SQL. Ces requêtes sont traduites en Aldat (le language natif de programmation de JRelix) et peut, de façon interactive, effectuer mises à jour, suppressions et sélections sur les différentes relations. La commande SQLSTMT permet d'exécuter des requêtes entièrement exprimées en SQL alors que la commande SQLEXP retourne au système une expression relationnelle qui peut interagir avec d'autres expressions relationnelles exprimées en Aldat dans la même requête.

Acknowledgments

First and foremost, I would really like to thank my thesis supervisor Professor Tim MERRETT. First for giving me a whole different point of view on relational databases, then for giving me the opportunity to go further on this subject by having my research in this domain of interest and finally for being always present to motivate my work, to give me a noticeable financial support, to give valuable insight during the implementation and to carefully read my thesis.

All this work and all my education would not have been if it would not have been my parents, Modou KHAYA and Aminata NDOYE. Their love, support and confidence in me and my ambitions just were everything. I cannot be more grateful to the man that was, is and will always be my hero and role model and to the woman that sacrificed her body, her time and her carreer for me without any hesitation.

I wish to thank my sister, Amy KHAYA, who has always been there for me and who has always driven me in becoming a better student and a better person. I also wish to thank my brother-in-law, Oumar DIA, for being the older brother that I have always dreamed of. His support was crucial.

Last but not least, I owe a special thought to my grand-mother, Sabelle NIANG, for showing me that there are few things stronger than your will when you are really decided to do something and to my whole family, back in Senegal who have always been very supportive of my education.

Contents

1 Introduction **1**

1.1 SQL Generalities . . . 1
1.1.1 History . . . 1
1.1.2 Data Modification . . . 1
1.1.3 Selections . . . 3
1.1.4 Aggregations . . . 6
1.1.5 Relation creation and deletion . . . 8
1.1.6 Triggers . . . 9
1.2 Aldat Generalities . . . 9
1.2.1 History . . . 9
1.2.2 Data Modification . . . 10
1.2.3 Selections . . . 11
1.2.4 Aggregations through Domain Algebra . . . 14
1.2.5 Relation creation and deletion . . . 16
1.2.6 Triggers . . . 17
1.3 Motivation and notions used in our research . . . 17
1.3.1 Motivation . . . 17
1.3.2 Context-free grammars and LL parsers . . . 19
1.3.3 Boolean Logic . . . 19
1.3.4 Abstract Syntax Trees . . . 20
1.4 General discussion about SQL and Aldat . . . 21
1.4.1 Quick overview of the main versions of SQL . . . 21
1.4.2 Quick overview of some of the RDBMS most used currently . . . 23
1.4.3 SQL-Aldat Comparison . . . 25

2 User's Manual 28
2.1 SQLSTMT command . . . 28
2.1.1 Selects . . . 28
2.1.2 Joins . . . 31
2.1.3 Aggregations . . . 34
2.1.4 Delete . . . 36
2.1.5 Update . . . 36
2.1.6 Multiple statements . . . 36
2.1.7 Omissions . . . 37
2.2 SQLEXP keyword . . . 37
2.2.1 Selects . . . 37
2.2.2 Joins . . . 38
2.2.3 Aggregations . . . 39
2.2.4 Multiple statements . . . 39
2.3 FILE Keyword . . . 40

3 Implementation details 42
3.1 SQLSTMT Command . . . 42
3.1.1 Selections . . . 42
3.1.2 Joins . . . 47
3.1.3 Aggregations . . . 54
3.1.4 Delete . . . 57
3.1.5 Update . . . 58
3.1.6 Insert . . . 59
3.1.7 General algorithm . . . 59
3.2 SQLEXP Keyword . . . 63
3.2.1 Selections . . . 63
3.2.2 General algorithm . . . 65
3.3 FILE Keyword . . . 66
3.3.1 General algorithms . . . 68
3.4 Commands grammar used which ANTLR cannot parse . . . 68
3.5 JRelix Configuration . . . 69
3.6 Tools . . . 69
3.6.1 ANTLR v2.7.7 and how it operates . . . 69
3.6.2 Files generated . . . 71
3.6.3 Oracle 7 SQL grammar from ANTLR website . . . 72
3.6.4 Boolean Parser . . . 72

4 Conclusion **76**
4.1 Recapitulation . . . 76
4.2 Future work . . . 77

A SQL Grammar **81**

B Sql Token Types **94**

Chapter 1

Introduction

1.1 SQL Generalities

We will narrow our overview of SQL to the areas we covered in our research. We will also use the SQL conventions of Oracle 7.

1.1.1 History

Based on the model defined by Dr. E.F. Codd in his paper [4], IBM developed in the 1970's, the Relational Database Project System R and the Structured English QUEry Language (SEQUEL)[2] used to perform queries on that database. SEQUEL became the Structured Query Language (SQL) after standardization. It has been extended by each Relational DataBase Management System (RDBMS) vendor. Oracle was the first to introduce SQL commercial implementation, in 1979.

1.1.2 Data Modification

Manipulating the data to reflect changes in the database is as important as viewing it, if not more important. So we will begin by the manipulation. This is mainly achieved through the three following commands in SQL : *Insert, Update and Delete*.

Update

Updates are expressed in SQL using an optional condition in the *Where* clause to "flag" the tuples that should be updated otherwise all tuples are updated. The syntax is the following :

> UPDATE *relation* SET *att* = *value*|*exp1* WHERE *condition-list* ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation from which tuples will be updated.
- *att* : The name of the attribute to update in the relation cited above.
- *value*|*exp1* : The new value of the attribute. It can be a constant, an attribute or an arithmetic expression involving the attributes of the relation.
- *condition-list* : The boolean expression corresponding to the conditions that need to be satisfied to have tuples flagged for the update. This boolean expression is expressed in the boolean algebra using the classic **(and, or, not)** sufficient set of operators.

Delete

Deletions are expressed in SQL using an optional condition in the *Where* clause to "flag" the tuples that should be deleted otherwise all tuples are deleted. That may be useful in order to keep the structure of the relation without the data. The syntax is the following :

> DELETE FROM *relation* WHERE *condition-list* ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation from which tuples will be deleted.
- *condition-list* : The boolean expression corresponding to the conditions that need to be satisfied to have tuples flagged for the delete.

Insert

In SQL inserting tuples in an existing relation can be done using the *Insert Into* command. A single tuple can be inserted using the keyword *Values* or another relation using a subquery. Nevertheless the subquery is not inserted as a relation but rather as a set of tuples. The syntax is the following :

> INSERT INTO *relation(att1, ..., attn)* VALUES *(value1, ..., valuen)* ;
> or
> INSERT INTO *relation(att1, ..., attn)* (*subquery*) ;

The definitions of the terms used are as follows :

- *relation(att1, ..., attn)* : The name of the relation in which tuples will be inserted. The attributes cited will be filled with the corresponding values. The other attributes of the relation, if any, are filled by nulls.
- *(value1, ..., valuen)* : The constants to insert in the relation. Their count should be the same as the cited attributes count from the relation.
- *subquery* : The subquery that will produce the set of tuples to insert. The count of attributes of that set should be the same as the cited attributes count from the relation.

1.1.3 Selections

The selectors in their different variants allow us to extract and view the elements of data that the user wants to see from the database. They are the commands that probably are the most used in a first approach to SQL.

Selects

Concerning the syntax of selections they accept one of these two options:

- *Distinct* : This option causes a selection to consider only distinct tuples.
- *All* : This option causes a selection to consider all tuples including duplicates. This is the default option in SQL.

The syntax is the following :

> SELECT [DISTINCT|ALL] *exp1* [[AS]*Alias1*], ..., *expn* [[AS]*Aliasn*]
> FROM *relation-list* WHERE *condition-list* ;

The definitions of the terms used are as follows :

- *exp1* [[AS]*Alias1*], ..., *expn* [[AS]*Aliasn*] : The expressions returned by the selection with their corresponding alias, which are optional. They can be a constant, an attribute or an arithmetic expression involving the attributes of the relation.
- *relation-list* : The name of the relations involved in the query.
- *condition-list* : The boolean expression corresponding to the conditions that need to be satisfied to have tuples returned in the selection.

It is possible to select all available fields from the relation using the "*" special character. The syntax is the following :

> SELECT [DISTINCT|ALL] * FROM *relation-list* WHERE *condition-list* ;

It is also possible to create a relation having the same structure as the source relation and insert the tuples at the same time using the *Select Into* command. It acts as the *Insert Into* command with a subquery. The syntax is the following :

> SELECT [DISTINCT|ALL] *exp1* [AS] [*Alias1*], ..., *expn* [AS] [*Aliasn*] INTO *relation* FROM *relation-list* WHERE *condition-list* ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation that will be created and in which tuples will be inserted.

Selections with Joins

The join conditions are specified in the *condition-list* of the *Where* clause. The relations involved are in the *relation-list* in the *from* clause. If no join condition is specified a cartesian product is made. The different joins that can be made are :

- **INNER JOIN**
  Specify the attribute from each relation on which you want to perform the join.

  SELECT X, Y, Z FROM R, S WHERE R.Y = S.Y ;

- **LEFT OUTER JOIN**
  To allow nulls on the right side of the join use (+) on that side.

  SELECT X, Y, Z FROM R, S WHERE R.Y = S.Y (+);

- **RIGHT OUTER JOIN**
  To allow nulls on the left side of the join use (+) on that side.

  SELECT X, Y, Z FROM R, S WHERE R.Y (+) = S.Y ;

- **FULL OUTER JOIN**
  You are obliged as mentioned below to make the union of the right and the left outer join to get the nulls on both side. The following is not allowed :

  SELECT X, Y, Z FROM R, S WHERE R.Y (+) = S.Y (+);

- **CARTESIAN PRODUCT**
  No condition is specified to perform directly the cartesian product.

  SELECT X, Y, Z FROM R, S;

Note that selection conditions as well as join conditions can be expressed at the same time in the *condition-list* as in the following :

SELECT X, Y, Z FROM R, S WHERE R.Y = S.Y AND W = 1;

Selections with Set Operators

Oracle allows the use of the following set operators:

- UNION
- UNION ALL (allows duplication in the union)
- INTERSECT
- MINUS

Nevertheless these are only set operators and may need to be combined with join operators to express complex joins.
For example, if we take the case of the *full outer join*:

```
(SELECT X, Y, Z FROM R, S WHERE R.Y (+) = S.Y)
UNION
(SELECT X, Y, Z FROM R, S WHERE R.Y = S.Y (+));
```

These set operators can only be used on *selections* and not directly on relations because there is a syntactic restriction. So the following is not allowed :

```
SELECT X, Y, Z FROM R UNION S;
```

1.1.4 Aggregations

Aggregation is performed in SQL through the aggregation functions such as *Sum or Max*. The optional presence of the *Group By* clause allows us to aggregate regarding certain criteria. For aggregated data an eventual condition will be expressed by a *Having* clause. So there is a clear distinction between the *Where* keyword reserved for individual tuples and the *Having* keyword reserved for aggregated tuples.

Concerning the syntax of these aggregation functions they accept one of these two options:

- *Distinct* : This option causes an aggregation function to consider only distinct values of the argument throughout all the tuples returned.
- *All* : This option causes an aggregation function to consider all values including all duplicates. This is the default option in SQL.

The syntax of the main aggregation functions is the following :

- SUM([DISTINCT|ALL] att) returns the sum of att.
- AVG([DISTINCT|ALL] att) returns the average of att.
- MIN([DISTINCT|ALL] att) returns the min of att.
- MAX([DISTINCT|ALL] att) returns the max of att.
- COUNT(*|[DISTINCT|ALL] att) returns the numbers of tuples. If * is specified all tuples will be counted otherwise tuples containing non-null values in att will be counted.

All aggregation functions except COUNT(*) ignore nulls.

In a complete query it will be in the following form :

> SELECT [DISTINCT|ALL] *aggregate-function*([DISTINCT|ALL]*exp1*) [AS] [*Alias1*], ..., *aggregate-function*(*expn*) [AS] [*Aliasn*], *exp1'* [AS] [*Alias1'*], ..., *expn'* [AS] [*Aliasn'*] FROM *relation-list* WHERE *condition-list* GROUP BY *exp1'*, ... , *expn'* HAVING *condition-list'*;

For example we could have the following :

> SELECT SUM(salary) AS SumSal, MAX(salary), department FROM employees WHERE seniority > 3 GROUP BY department HAVING SUM(salary) > 100000;

This would give the total amount of salaries and the greatest salary by department for the employees having more than 3 years of seniority where the total amount of salaries of the department is greater than 100000. The *Having* condition applies to the total returned.

Note that when no alias is specified a system-generated name is used and that the system does not allow us to reuse the alias SumSal in the *Having* clause. That is allowed in MySQL only. The definitions of the terms used are as follows :

- *aggregate-function*(*exp1*) [AS] [*Alias1*], ..., *aggregate-function*(*expn*) [AS] [*Aliasn*] : The expressions on which the aggregation functions will be performed and their corresponding aliases, if any.

- *relation-list* : The name of the relations involved in the query.
- *condition-list* : The boolean expression corresponding to the conditions that need to be satisfied to have simple tuples considered for the aggregation.
- *condition-list'* : The boolean expression corresponding to the conditions that need to be satisfied by the aggregated tuples to be returned in the selection.
- *exp1'* [AS] [*Alias1'*], ..., *expn'* [AS] [*Aliasn'*] : The expressions on which the aggregation will be based and their corresponding aliases, if any.

1.1.5 Relation creation and deletion

In SQL a relation and its attributes creations are linked. It is in the relation creation command that the the attributes datatypes and properties are specified. The syntax is the following :

CREATE TABLE *relation* (*col1 datatype1*, ..., *coln datatypen*) ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation that will be created.
- *col1 datatype1*, ..., *coln datatypen* : The name of the attributes included in this relation and their corresponding datatypes.

To delete a relation you just use the following syntax :

DROP TABLE *relation* ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation that will be deleted.

1.1.6 Triggers

Triggers can be used in Oracle 7. They can be triggered by *Update, Delete or Insert* statements. The trigger body can be executed *Before or After* the data manipulation. The old and new configuration of the relation are referenced as *OLD* and *NEW*. The trigger can be started only once for the statement or for each tuple affected by the statement using the *FOR EACH ROW* command.

We could have as an example :

```
CREATE TRIGGER trig
AFTER UPDATE ON Employees
FOR EACH ROW
WHEN NEW.hierarchy = 'Manager'
BEGIN
INSERT INTO Managers VALUES (New.fName, New.lName)
END
```

Recursive Triggers can be implemented but there is a hard coded limit of 50 recursive calls before raising an error to prevent infinite loops. This limit will be configurable after *Oracle 8* which allows a mechanism to prevent infinite loops and have the flexibility of choosing the limit to reach.

1.2 Aldat Generalities

We will restrict our overview of Aldat to the areas we covered in our research. We will use a similar breakdown for this section to the breakdown in the previous one.

1.2.1 History

Based on the same paper from Codd [4] McGill began the *Al*gebraic approach to *dat*a (Aldat) Project in the 1970's and extended Codd's Relational Algebra to the Domain Algebra proposed by Merrett [12]. This extension toward the Domain Algebra allowed Aldat to directly deal with Programming aspects in its core from a relational point of view.

The latest implementation of Aldat as a RDBMS is the *J*ava implementation of a *Rel*ational database programming language in Un*ix* (JRelix). This

implementation is principally based on the contributions made by Baker[1], Yuan[28] and Hao[9], through their M.Sc. theses and projects at McGill.

1.2.2 Data Modification

Update

Updates are expressed in Aldat using a join to "flag" the tuples that should be updated otherwise all tuples are updated. The syntax is the following :

> UPDATE *relation* CHANGE *att* <- *value*|*exp1* USING *relational expression* ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation from which tuples will be updated.
- *att* : The name of the attribute to update in the relation cited above.
- *value*|*exp1* : The new value of the attribute. It can be a constant, an attribute or an arithmetic expression involving the attributes of the relation.
- *relational expression* : The relational expression that is joined with *relation* to flag its tuples for the update.

Another alternative is to use conditions rather than a join to update. In that case there is no join to "flag" the tuples that should be updated so all of them are considered for the update and it is the conditional expression that changes only the tuples that need to. Of course this is not optimal because even though only one tuple may have to be updated all of them will be selected for the update. The syntax is the following :

> UPDATE *relation* CHANGE *att* <- IF *condition-list* THEN *value*|*exp1* ELSE *att* ;

The definitions of the terms used are as follows :

- *condition-list* : The boolean expression corresponding to the conditions that need to be satisfied to do the update.

Delete

Deletions are expressed in Aldat using a keyword to specify the tuples that should be deleted. The syntax is the following :

UPDATE *relation* DELETE *relational expression* ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation from which tuples will be deleted.
- *relational expression* : The relational expression that is used to specify the deletion.

Insert

Insertions are performed in Aldat using the following syntax :

UPDATE *relation* ADD *relational expression* ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation to which tuples will be added.
- *relational expression* : The relational expression whose tuples will be inserted in *relation*.

1.2.3 Selections

Selects

In Aldat selections are divided into three categories :

- Projections consist of a selection of available attributes but with all tuples returned. They use the following syntax :

  [*att1*, ..., *attn*] IN *relational expression* ;

  The definitions of the terms used are as follows :

  - *att1*, ..., *attn* : The attributes returned by the selection.

  - *relational expression* : The relational expression that is used as a source for the selection.
  - *condition-list* : The boolean expression corresponding to the conditions that need to be satisfied to have tuples returned in the selection.

- Selections consist of a selection of available tuples but with all attributes returned. They use the following syntax :

  WHERE *condition-list* IN *relational expression* ;

  The definitions of the terms used are as follows :

  - *relational expression* : The relational expression that is used as a source for the selection.
  - *condition-list* : The boolean expression corresponding to the conditions that need to be satisfied to have tuples returned in the selection.

- T-Selections consist first of a selection of available tuples and then of available attributes. It is a fusion of the two preceding selections. They use the following syntax :

  [*att1*, ..., *attn*] WHERE *condition-list* IN *relational expression* ;

  The definitions of the terms used are as follows :

  - *att1*, ..., *attn* : The attributes returned by the selection.
  - *relational expression* : The relational expression that is used as a source for the selection.
  - *condition-list* : The boolean expression corresponding to the conditions that need to be satisfied to have tuples returned in the selection.

From this point we will be considering T-Selections throughout this section since they can express anything that can be expressed with the two others. Also note that Aldat will always return distinct tuples, being fully relational.

Selections with Joins

Aldat being all relational by design there are no set operators but only join operators. They are :

- **INNER JOIN**
  You specify the attribute from each relation on which you want to perform the join.

  [X, Y, Z] IN (R[Y:IJOIN:Y]S) ;

  If the attribute is the same on both relations then you are not obliged to specify it and Aldat will perform the join on the common attribute available :

  [X, Y, Z] IN (R IJOIN S) ;

  This can be performed for all joins. From now on we will be considering the case where the join attributes are explicit.

- **LEFT OUTER JOIN**
  You specify that you want to allow nulls on the right side of the join using the *ljoin* operator.

  [X, Y, Z] IN (R[Y:LJOIN:Y]S);

- **RIGHT OUTER JOIN**
  You specify that you want to allow nulls on the left side of the join using the *rjoin* operator.

  [X, Y, Z] IN (R[Y:RJOIN:Y]S) ;

- **FULL OUTER JOIN**
  You specify that you want to allow nulls on both sides of the join using the *ujoin* operator.

  [X, Y, Z] IN (R[Y:UJOIN:Y]S) ;

- **DIFFERENCE JOIN**
  You specify that you do not want to allow the tuples matching on the right side of the join using the *djoin* operator.

  [X, Y, Z] IN (R[Y:DJOIN:Y]S) ;

- **DIFFERENCE RIGHT JOIN**
  You specify that you do not want to allow the tuples matching on the left side of the join using the *drjoin* operator.

  [X, Y, Z] IN (R[Y:DRJOIN:Y]S) ;

- **SYMMETRIC DIFFERENCE JOIN**
  You specify that you do not want to allow the tuples matching on both sides of the join using the *sjoin* operator.

  [X, Y, Z] IN (R[Y:SJOIN:Y]S) ;

- **CARTESIAN PRODUCT**
  If there is no common attributes the cartesian product is directly performed using the *ijoin* operator.

  [X, Y, Z] IN (R IJOIN S) ;

  Note that selection conditions could be used on the relational expression resulting from the join as in the following :

  [X, Y, Z] WHERE W = 1 IN (R[Y:IJOIN:Y]S) ;

1.2.4 Aggregations through Domain Algebra

We will limit our overview of the Domain Algebra to the elements relevant to the aggregation. The domain algebra can be divided into two main types of operations :

- The horizontal operations that cover arithmetic expressions including attributes, the use of constants and the renaming (aliasing) of the attributes.

- The vertical (aggregations) operations : *reduction*, *equivalence reduction*, *functional mapping* and *partial functional mapping*.

We will principally focus on the latter ones. The first operation relevant is the *reduction*. The syntax is the following :

LET *Alias* BE RED *Operator* OF *exp* ;

The definitions of the terms used are as follows :

- *Alias* : The name of the aggregated expression.
- *Operator* : One of the operator that can be used in reduction operations : "+", "*", "max", "min", "and", "or".
- *exp* : The expression on which the aggregation will be performed.

The *reduction* allows us to make aggregation but does not do it using a grouping. For that we have to use the *equivalence reduction* whose syntax is the following :

LET *Alias* BE EQUIV *Operator* OF *exp* BY *att1, ..., attn*;

The definitions of the terms used are as follows :

- *Alias* : The name of the aggregated expression.
- *Operator* : One of the operator that can be used in reduction operations : "+", "*", "max", "min", "and", "or".
- *exp* : The expression on which the aggregation will be performed.
- *att1, ..., attn* : The attributes on which the grouping will be performed.

Aldat also allows us to perform operations in a cumulative manner. The *functional mapping* is used for that with the following syntax :

LET *Alias* BE FUN *Operator* OF *exp* ORDER *att1, ..., attn*;

The definitions of the terms used are as follows :

- *Alias* : The name of the aggregated expression.
- *Operator* : One of the operator that can be used in functional mapping operations : "+", "*", "-", "", "max", "min", "||", "pred", "succ", "and", "or".
- *exp* : The expression on which the aggregation will be performed.
- *att1, ..., attn* : The attributes that are the basis of the accumulation.

The accumulation can also be performed under a grouping with the following syntax :

LET *Alias* BE PAR *Operator* OF *exp* ORDER *att1, ..., attn* BY *att1', ..., attn'*;

The definitions of the terms used are as follows :

- *Alias* : The name of the aggregated expression.
- *Operator* : One of the operator that can be used in functional mapping operations : "+", "*", "-", "", "max", "min", "||", "pred", "succ", "and", "or".
- *exp* : The expression on which the aggregation will be performed.
- *att1, ..., attn* : The attributes that are the basis of the accumulation.
- *att1', ..., attn'* : The attributes on which the grouping will be performed.

1.2.5 Relation creation and deletion

In Aldat a relation and its attributes creations are totally independent. The attributes and their datatypes have to be created first and then the relation can be created. The syntax is the following :

DOMAIN *col1 datatype1*;
...
DOMAIN *col1 datatype1*;
RELATION *relation* (*col1*, ..., *coln*) ;

The definitions of the terms used are as follows :

- *relation* : The name of the relation that will be deleted.
- *col1 datatype1*, ..., *coln datatypen* : The name of the attributes included in this relation and their corresponding datatypes.

To delete a relation and the attributes composing it you just use the following syntax :

DR *relation* ;
DD *col1*, ..., *coln*;

The definitions of the terms used are as follows :

- *relation* : The name of the relation that will be deleted.
- *col1*, ..., *coln* : The name of the attributes that will be deleted.

1.2.6 Triggers

Triggers called event handlers are also part of Aldat. They can be triggered by *Change, Delete or Add* statements. The trigger body can be executed before (*Pre*) or after (*Post*) the data manipulation. In the case of a *Pre* The old configuration of the relation is referenced as the relation itself and the new configuration as *TRIGGER*. In the case of a *Post* it is the other way around. A simple example would be :

Comp:Post:Change:Employees() is
Update Managers Add ([fName, lName] where hierarchy = "Manager" in Employees);

Recursive Triggers are allowed without any limit of recursion. It is the responsibility of the programmer to avoid infinite loops. This is another characteristic of Aldat: there are no restrictions to give the programmer the full control of the tools Aldat provides but it is its responsibility to make sure that he uses them right.

1.3 Motivation and notions used in our research

1.3.1 Motivation

SQL being the de facto standard query language for databases, an SQL interface was obviously missing from the Aldat and JRelix Project at McGill

which have deep insights in the Relational and Domain Algebras. Pretty much any existing system supporting queries would provide an SQL interface to it and the absence of an SQL front-end to this system would definitely lower its popularity despite the advanced capabilities of Aldat.

The best way to show that Aldat subsumes SQL is by providing an SQL-Aldat translation. Therefore the interface would then just be the translator that would feed the system with the translated statements to execute them. There would be no need to make an interface that would directly deal with the internals of the system as it would be the case if Aldat did not subsumes SQL.

Even an incomplete translator at first, that could be completed in future research, is a proof of concept that Aldat does subsume SQL and can provide a translator as an interface. This challenge was so appealing that the decision was made to start a research focusing on the design and the implementation of an extension to JRelix that would provide an SQL interface to it. It would allow JRelix to be available for several purposes that would not involve directly the learning of Aldat, e.g. being plugged to existing systems with an SQL interface, while being still available for more complex purposes that could be achieved through Aldat.

This research will not cover the whole scope of SQL but would provide a strong basis of the translation between SQL and Aldat that can be extended to have a full coverage of the SQL language in a future work. It has been principally based on the the following key notions described with more details in the following sections and which usage is very common in the Computer Science environment:

- Context-free grammars and LL parsers
- Boolean Logic
- Abstract Syntax Trees

These three sections correspond to the three main steps of the translation :

- The SQL statements are parsed by the LL parser generated from the SQL context-free grammar
- They are further parsed by our own boolean parser using the Boolean Logic

- Abstract Syntax Trees are then generated from these statements to translate them in Aldat

1.3.2 Context-free grammars and LL parsers

Context-free grammars, which have been first formalized by N. Chomsky [3], play a major role in Computer Science because they can describe the design of most of the currently used programming languages and compilers. They can almost always be described by a context free grammar because it allows us to capture the block structure inherent to most programming languages nowadays. The main other advantage of context-free grammars is that parsing algorithms can generally be found or generated rather easily from them.

LL parsers cannot parse all the context-free grammars but the subset of them whose rules can be applied from *left to right* i.e. using a *leftmost derivation* from the root of the grammar. This is often the case in Computer Science because the programming languages are often written by people used to read and write from left to right. Therefore they can be described by the LL grammars and parsed by the corresponding parsers.

In the case of our research we used a context-free grammar[26] expressed using the Extended BackusNaur Form notation (EBNF)[27] to generate a parser with a meta parser, ANTLR [24], which is LL. We choose the ANTLR Metaparser because we could easily find an SQL grammar for it and the parser it generates can also generate ASTs[1] after the parsing. JavaCC[15] is another widely used Meta Parser and actually is the one used in JRelix to generate its parser. Nevertheless we could not find a simple SQL grammar for it so we chose ANTLR.

1.3.3 Boolean Logic

Boolean Logic is one of the most used notions in Computer Science because it allows us to describe computational logic. Its scope goes beyond computers and Boolean Logic is also used in electrical engineering to describe the electronic circuits using the gates as operators.

It is Boolean Logic which is used in both SQL and Aldat to express the conditions, e.g. to "flag" tuples for a selection. This was a critical notion

[1]See Section 1.3.4

for us because SQL expresses selection conditions and join conditions in the same clause which is not the case in Aldat and we had to separate them by breaking down the clause into simple elements and recombining each boolean subexpression to express the same statement. This would not have been possible without Boolean Logic.

Specially considering that the parser simply copies the content of the *Where* clause in the AST, we had to design and implement our own Boolean Parser [2] that would further process the ASTs generated by the parser to have the *Where* clause broken down using Boolean Logic to simple boolean expressions. We consider the sufficient set {AND, OR, NOT} of operators and used the parentheses to disambiguate the boolean expressions by imposing the evaluation order of the different clauses composing it.

1.3.4 Abstract Syntax Trees

A powerful intermediate form

Abstract Syntax Trees (AST) are a very powerful intermediate form between a source code written in a programming language, in our case the SQL statements, and generally the binary code or any other language corresponding to the result of the grammar, in our case the translated Aldat statements. It is often used for optimization purposes because it allows us to keep the structure of the program without being sensitive to the the grammar used to parse it as in the case of a Parse Tree. This is made possible by the Abstract Syntax that specifies the range of possible structures of the language. This Abstract Syntax may be defined using the Abstract Syntax Notation One (ASN.1) [20]. The usefulness of the ASTs have already been recognized [25][7][11]. The JRelix implementation actually utilize the ASTs as an intermediate form after the parsing for the interpreter.

Tree structure of an AST

An AST consists of a finite tree where interior nodes are operators of the language and their children the operands. The leaves are then variables or constants used in that statement. So the statement which will be broken down from the root to the leaves capturing its structure in the tree structure.

[2]See Section 3.6.4

The translation of the statement is then eased by the fact that its structure is already known.

ASTs in the context of our translator

- Ordered Trees
  In the case of our implementation, as it is often the case with ASTs actually, the ASTs generated are ordered. The ordering of these ASTs is important in the sense that the order of the clauses of a statement has to be kept in its intermediate form and is an integral part of the structure of that statement.Therefore it would not have been logical to generate unordered trees.

- Boolean Parser
  In the case of the *Where* clause the AST generated is flat. So we had to develop our own Boolean Parser and we would then generate ASTs with the same structure as the ones generated by the ANTLR parser and replace the flat one by our own. From there we can translate the whole SQL statements using the ASTs.

1.4 General discussion about SQL and Aldat

1.4.1 Quick overview of the main versions of SQL

- **SQL-86** [19]
  This is the first standardization of SQL adopted by the American National Standards Institute (ANSI) in 1986 and the International Standards Organization (ISO) in 1987. It has been slightly revised and adopted by the U.S. National Institute of Standards and Technology (NIST) in 1989 to become SQL-89 (also known as SQL1 or FIPS 127-1). A conformance testing suite [17] to the standard has been provided by NIST to improve the standardization process for the RDBMS vendors.

- **SQL-92 (SQL2)** [21]
  This is the first major revision of the SQL-86 standard. There was a clear effort made to go from SQL-86 with a quite incomplete standardization of the language with an important part of the scope left as

"implementor-defined" to a full coverage of the standard. It standardized the ANSI JOIN notation, between many other standardizations, and introduced the levels of compliance to the standard notion. There were three levels of compliance defined in SQL-92 :

  - Entry-level conformance (only the barest improvements to SQL-89)
  - Intermediate-level conformance (a generally achievable set of major advancements)
  - Full conformance (total compliance with the SQL-92 features)

  The entry level is also known as FIPS 127-2 and has a conformance testing suite available provided by NIST.

- **SQL-99 (SQL3)** [22]
  This is the second major revision of the SQL-86 standard. It mainly added the following features in the standardization : regular expression matching, recursive queries and triggers. It also standardized the support for procedural and control-of-flow statements with the definition of a standard procedural programming language extension which has been called SQL/Persistent Stored Module (SQL/PSM). The new level of compliance defined are : Core SQL99 and Enhanced SQL99. They are no longer certified by NIST since 1996 but by the vendors themselves.

- **SQL-2003** [23]
  This revision mainly introduced XML and more generally semi-structured data support in SQL. This has been a major awaited improvement [5] with the emergence of XML, and other forms of semi-structured data, in the late 1990's as one of the most reliable ways to transfer large amount of data. SQL-2003 standardized [6] the mapping between the two languages.

1.4.2 Quick overview of some of the RDBMS most used currently

Oracle

Oracle [18] is a leader among RDBMS vendors. That is one of the reason we chose an Oracle SQL grammar for our research. The main characteristics specific to this vendor that we will deal with in our research are :

- **The default column expression name**
  The expression itself is used as a name. That can be done using quoted identifiers which allows all characters in their composition.

- **The column aliases references**
  The column aliases can not be referred to in the *Where* and *having* clause.

- **Old outer join notation**
  The outer join notation prior to SQL-92 is the following :

  SELECT X, Y, Z FROM R, S WHERE R.Y = S.Y (+);

- **Procedural programming language extension** : Procedural Language/SQL (PL/SQL)

MS SQL Server

Microsoft [14] has also an important share of the RDBMS world because of its high compatibility with other Microsoft software that are well spread, e.g. its operating system Windows. The main characteristics specific to this vendor that are relevant to our research are :

- **The default column expression name**
  "No Name" is used as a name.

- **The column aliases references**
  The column aliases can not be referred to in the *Where* and *having* clause.

- **Old outer join notation**
  The outer join notation prior to SQL-92 is the following :

SELECT X, Y, Z FROM R, S WHERE R.Y *= S.Y ;

- **Procedural programming language extension** : Transact-SQL(T-SQL)

IBM DB2

IBM [10] was the precursor in the research on SQL but has been overtaken by Oracle in its commercial implementation. The main characteristics specific to this vendor that are relevant to our research are :

- **The default column expression name**
  The column number is used as a name.
- **The column aliases references**
  The column aliases can not be referred to in the *Where* and *having* clause.
- **Old outer join notation**
  The outer join was not available prior to SQL-92 so it is the only notation available.
- **Procedural programming language extension** : SQL Procedural Language (SQL PL)

MySQL

MySQL [16] is one of the most known Open Source RDBMS. The main characteristics specific to this vendor that are relevant to our research are :

- **The default column expression name**
  The expression itself is used as a name.
- **The column aliases references**
  The column aliases can not be referred to in the *Where* but can be used in the *having* clause.
- **Old outer join notation**
  The outer join was not available prior to SQL-92 so it is the only notation available.

- **Procedural programming language extension** : MySQL

1.4.3 SQL-Aldat Comparison

Joins

SQL unlike Aldat can be very ambiguous with joins. It begins the ambiguity directly by using the *where* keyword which is also the same to express a condition. Furthermore some database systems have adopted the ANSI (American National Standards Institute) join keywords ("INNER JOIN", "OUTER JOIN", ...), included in SQL-92, only very recently which leaves a certain amount of older systems still running without the ANSI joins. Oracle only adopted the ANSI joins with the version 9. Since our translator is based on Oracle 7 [18] grammar we will not be able to use the ANSI joins.

Aggregations

The aggregations performed in SQL through the aggregation functions are a subset of the domain algebra defined in Aldat which allows us to do as much and even more being more generic. Also the distinction between the *Where* keyword reserved for individual tuples and the *Having* keyword reserved for aggregated tuples in SQL is nonexistent in Aldat in which the condition for aggregated data as much as individual data will be expressed by the *Where* keyword.

Furthermore the domain algebra allows us to use alias as for aggregated expressions in any part of a statement whereas in SQL for example column aliases can not be used in the *Having* clause because of the evaluation order of the clauses, in particular the *Having* clause is evaluated before the *Select* clause, apart from in MySQL where there is a different evaluation order which allows us to use column aliases in the *Having* clause.

Aldat, having a totally relational design, considers only distinct values. So the default option is the DISTINCT in the aggregation functions. Once again the difference in the design of the the two languages resurfaces. Nevertheless it is possible to express aggregations on duplicates.

Data Modification

- **Inserts**
  In SQL inserting tuples is done considering the set of rows to insert whereas in Aldat the relation to insert itself is considered as a whole and added directly.

- **Updates and Deletions**
  Updates and Deletions are expressed in SQL with a condition in the *Where* clause to "flag" the tuples that should be updated or deleted. *Updates* are expressed in Aldat with a join on the relation specified in the *Using* clause to "flag" the tuples that should be updated. Relations in Aldat are considered as a whole whereas in SQL it will rather be individual tuples. This is another design decision which emphasizes this difference between Aldat and SQL.

Relation creation and deletion

In SQL the attributes are bound to the relation incorporating them and so their creation and deletion are linked to those of the relation. Whereas in Aldat the relation only incorporates the attributes in its declaration. The attributes were created before the relation even existed and will continue to exist even after the deletion of the relation. The only restriction that exists is the deletion of an attribute that is still used in a relation which is forbidden because it would break the structure of that relation.

A further difference is due once again to the Domain Algebra notion which is absent in SQL and an important part of Aldat conception. SQL only incorporates the Relational Algebra notion and uses functions, e.g. aggregation functions, to deal with operations on attributes that should be performed using the Domain Algebra.

These differences are part of the overall difference between Aldat and SQL concerning the principles of abstraction and closure [13]. Aldat respects them whereas SQL does not. The principle of closure stipulates that operations result should conserve the type of the operands, e.g. operations on relations produce relations. And the principle of abstraction stipulates that operations should be independent of the structure of the operands and of their context. Another example of the application of this latter principle is the case of the virtual attribute notion that Aldat integrates and which is absent from SQL.

A straight-forward example of the first is the boundless recursive definition of a view that can be done in Aldat and can not in SQL.

Triggers

Triggers body have to be written in PL/SQL, *Begin ... End* block, to deal with the programming aspect of triggers which embodies one of the main advantages of Aldat. The language itself have been designed to include both relational and programming aspects of databases. Whereas here we have a difference between the SQL which is intended for and only for the relational aspect and the PL/SQL which adds the programming aspect. One of the main problems of this differentiation apart from the obvious efficiency one and the lack of uniformity is that PL/SQL is an extension to SQL made by Oracle and so it can not be used with other databases such as MS SQL Server for example which uses its own extension T-SQL. So the portability of the code, which was not totally assured because of the differences noted before in the notations, is definitely lost when you introduce the programming side.

Chapter 2

User's Manual

To get started let's first start JRelix in which our translator is integrated eventually. First of all let's open a terminal with a command line. Let's type in the command line relix to launch it :

$ **relix**

Which will output :

```
Starting stand alone JRelix.

+----------------------------------------+
|          Relix Java version 0.96       |
| Copyright (c) 1997 – 2007 Aldat Lab    |
|       School of Computer Science       |
|           McGill University            |
+----------------------------------------+
>
```

2.1 SQLSTMT command

2.1.1 Selects

Let's now consider consider simple selects :

- 1 First selecting an attribute from a relation :

Type in the command line the SQL query that we want to translate:
> sqlstmt "select a into destRel from b;";
Which will execute the corresponding Aldat query :

```
destRel<- [a] in b;
```

We specified the target relation because this is a command so it cannot have a void action. In the section 2.2 we will see how to have selections without it.

- 2 Then selecting a qualified attribute from a relation :

  > sqlstmt "select b.a into destRel from b;";

  ```
  destRel<- [a] in b;
  ```

- 3 Then selecting distinct elements of a attribute from a relation :

  > sqlstmt "select distinct a into destRel from b;";

  ```
  destRel<- [a] in b;
  ```

  Aldat is totally relational so the tuples returned are always distinct.

- 4 Then selecting all attributes from a relation :

  > sqlstmt "select * into destRel from b;";

  ```
  destRel<- b;
  ```

- 5 Now selecting attributes from a subquery with a where condition :

  > sqlstmt "select a, b into destRel from (select a,b,c from d where (e=1 and f='a') or g=2);";

  ```
  destRel<- [a,b] in ([a,b,c] where (e=1 and f="a") or
  g=2 in d);
  ```

- 6 Then selecting expression from a relation :

```
> sqlstmt "select a + b into destRel from c;";
let No_Name be a+b;
destRel<- [No_Name] in c;
```

We took the same naming convention as MS SQL Server concerning expressions without alias i.e. using *No Name*. Oracle and MySQL uses the expression itself, DB2 uses the column number.

- 7 Then selecting attribute with an alias from a relation :

```
> sqlstmt "select a as b into destRel from c;";
let b be a;
destRel<- [b] in c;
```

- 8 Then selecting attribute with an alias without the keyword *as* from a relation :

```
> sqlstmt "select a b into destRel from c;";
let b be a;
destRel<- [b] in c;
```

- 9 Then selecting attributes from a query with a *LIKE* condition :

```
> sqlstmt "select a into destRel from b where c like '%OMPUTE%';";
destRel<- [a] where att = "c" in (grep(att) "ompute"
in b);
```

- 10 Then selecting attributes from a query with a condition involving an expression:

```
> sqlstmt "select a into destRel from b where c + d = 1;";
destRel<- [a] where c+d=1 in b;
```

2.1.2 Joins

Inner Joins and Difference Join

Now we can get to more complex selects implying joins :

- 11 First the most straightforward select with join on one attribute :

  > sqlstmt "select b into destRel from c, d where c.a = d.a;";
  ```
  destRel<- [b] in (c[a:ijoin:a]d);
  ```

- 12 Now select with join on several attributes :

  > sqlstmt "select b into destRel from c, d where c.a = d.a and c.e = d.e and c.f = d.g;";
  ```
  destRel<- [b] in (c[a,e,f:ijoin:a,e,g]d);
  ```

- 13 Now select with a difference join on several attributes :

  > sqlstmt "select b into destRel from c, d where c.a = d.a and not(c.e = d.e and c.f = d.g);";
  ```
  destRel<- [b] in (c[a:ijoin:a]d) djoin (c[e,f:ijoin:e,g]d);
  ```

- 14 Now select with a join and a condition on several attributes :

  > sqlstmt "select b into destRel from c, d where c.a = d.a and e=1 and f='a';";
  ```
  destRel<- [b] where e=1 and f="a" in (c[a:ijoin:a]d);
  ```

Left Outer Join, Right Outer Join and Full Outer Join

We can then get to more subtle joins :

- 15 First a simple right outer join :

  > sqlstmt "select b into destRel from c, d where c.a (+) = d.a;";
  ```
  destRel<- [b] in (c[a:rjoin:a]d);
  ```

The "(+)" allows nulls on the left side of the join so that makes it a right outer join.

- 16 The corresponding left outer join :

  > sqlstmt "select b into destRel from c, d where c.a = d.a (+);";
  `destRel<- [b] in (c[a:ljoin:a]d);`

- 17 Now a little variant not allowed in Oracle SQL but that we allow to ease full outer joins expression :

  > sqlstmt "select b into destRel from c, d where c.a (+) = d.a (+);";
  `destRel<- [b] in (c[a:ujoin:a]d);`

- 18 Now a left outer join on several attributes :

  > sqlstmt "select b into destRel from c, d where c.a = d.a (+) and c.e = d.e (+);";
  `destRel<- [b] in (c[a,e:ljoin:a,e]d);`

- 19 Now select with 2 different joins on several attributes :

  > sqlstmt "select b into destRel from c, d where c.a = d.a (+) and c.e = d.e;";
  `destRel<- [b] in (c[a:ljoin:a]d) ijoin (c[e:ijoin:e]d);`

- 20 Now select with the 3 different joins on several attributes :

  > sqlstmt "select b into destRel from c, d where c.a = d.a (+) and c.e = d.e and c.f (+) = d.g;";
  `destRel<- [b] in (c[a:ljoin:a]d) ijoin (c[e:ijoin:e]d) ijoin (c[f:rjoin:g]d);`

- 21 Now select with 1 inner join and 2 left outer joins on several attributes :

> sqlstmt "select b into destRel from c, d where c.a = d.a
> and c.e = d.e (+) and c.f = d.g (+);";
> `destRel<- [b] in (c[a:ijoin:a]d) ijoin (c[e,f:ljoin:e,g]d);`

- 22 Now select with 1 inner join or 2 left outer joins on several attributes
  :

  > sqlstmt "select b into destRel from c, d where c.a = d.a or
  > c.e = d.e (+) and c.f = d.g (+);";
  > `destRel<- [b] in (c[a:ijoin:a]d) ujoin (c[e,f:ljoin:e,g]d);`

Joins with unqualified attributes or none

It is possible to use joins using the unqualified names of the attributes. The system will be able to differentiate them. For example if we consider a relation c with the attributes a,b and f and a relation d with the attribute e and no shared attributes between the 2 relations. We will have the following:

> 23 > sqlstmt "select b into destRel from c, d where c.a = c.f;";
> `destRel<- [b] where a=f in (c ijoin d);`

With the same output as :

> 24 > sqlstmt "select b into destRel from c, d where a = f;";
> `destRel<- [b] where a=f in (c ijoin d);`

and :

> 25 > sqlstmt "select b into destRel from c, d where c.a = d.e;";
> `destRel<- [b] in (c[a:ijoin:e]d);`

With the same output as :

> 26 > sqlstmt "select b into destRel from c, d where a = e;";
> `destRel<- [b] in (c[a:ijoin:e]d);`

and if no attribute is specified we have the cartesian product :

27 > sqlstmt "select b into destRel from c, d;";
```
destRel<- [b] in (c ijoin d);
```

2.1.3 Aggregations

Aggregations are an important way to summarize data. They can be used the following way :

- 28 First a simple sum without alias :

  > sqlstmt "select sum(salary) into destRel from employees;";
  ```
  let No_Name be red + of salary;
  destRel<- [No_Name] in employees);
  ```

- 29 Then a simple distinct sum without alias :

  > sqlstmt "select sum(distinct salary) into destRel from employees;";
  ```
  let No_Name be red max of (fun + of salary order salary);
  destRel<- [No_Name] in employees);
  ```

- 30 Then a simple avg without alias :

  > sqlstmt "select avg(salary) into destRel from employees;";
  ```
  let No_Name be red + of salary / red + of 1;
  destRel<- [No_Name] in employees);
  ```

- 31 Then a simple max without alias :

  > sqlstmt "select max(salary) into destRel from employees;";
  ```
  let No_Name be red max of salary;
  destRel<- [No_Name] in employees);
  ```

- 32 Then a simple min without alias :

> sqlstmt "select min(salary) into destRel from employees;";
```
let No_Name be red min of salary;
destRel<- [No_Name] in employees);
```

- 33 Then a simple count without alias :

  > sqlstmt "select count(*) into destRel from employees;";
  ```
  let No_Name be red + of 1;
  destRel<- [No_Name] in employees);
  ```

- 34 Then another simple count without alias :

  > sqlstmt "select count(salary) into destRel from employees;";
  ```
  let No_Name be red + of (if isnull(salary) then 0 else 1);
  destRel<- [No_Name] in employees);
  ```

  As usual in SQL the number of tuples having a non NULL value of the column.

- 35 Now a simple distinct count without alias :

  > sqlstmt "select count(distinct salary) into destRel from employees;";
  ```
  let No_Name be red max of fun + of 1 order (salary);
  destRel<- [No_Name] in employees);
  ```

- 36 Now a more complex sum involving an alias, a group by and a having condition :

  > sqlstmt "select sum(salary) S into destRel from employees group by dept having S > 10;";
  ```
  let S be equiv + of salary by dept;
  destRel<- [S] where S > 10 in salary;
  ```

As in MySQL we allow the column aliases to be used in the GROUP BY clause.

2.1.4 Delete

Deletions can be expressed the following way :

- 37 A delete without a condition :

  > sqlstmt "delete from b;";

  ```
  update b delete b;
  ```

- 38 A delete with a condition :

  > sqlstmt "delete from b where a = 1;";

  ```
  update b delete where a = 1 in b;
  ```

2.1.5 Update

Updates are also supported :

- 39 Updating a attribute with a condition on another :

  > sqlstmt "update a set b = 10 where c = 2;";

  ```
  update a change b <- if c = 2 then 10 else b;
  ```

2.1.6 Multiple statements

It is also possible to write several statements at once :

> sqlstmt "delete from b where a = 1;
select b into destRel from c, d where c.a = d.a or c.e = d.e (+)
and c.f = d.g (+);
update a set b = 10 where c = 2;";

```
update b delete where a = 1 in b;
destRel<- [b] in (c[a:ijoin:a]d) ujoin (c[e,f:ljoin:e,g]d);
update a change b <- if c = 2 then 10 else b;
```

2.1.7 Omissions

In the current version of the tool we do not support all the commands of SQL because the parser we chose does not support the full SQL grammar. The most important omissions are listed below:

- The create command : table, view, trigger, ... etc
- The drop command
- The insert command

2.2 SQLEXP keyword

We added the SQLEXP keyword to be able to use SQL relational expression in enclosing Aldat statements. Any Aldat statement using relational expression can use it.

2.2.1 Selects

Let's now consider Aldat statements involving the SQLEXP keyword:

- 1 printing a SQL relational expression:

  > pr sqlexp "select a from b;";
  > Which will execute Aldat query :

  ```
  pr ([a] in b);
  ```

- 1' Assigning a join between an existing relation and a SQL relational expression:

  > d<- c ijoin sqlexp "select a from b;";
  > Which will execute Aldat query :

  ```
  d<- c ijoin ([a] in b);
  ```

- 1" Printing a projection of a SQL relational expression:

> pr [a] in (sqlexp "select a, c from b;");
Which will execute Aldat query :
```
pr [a] in ([a, c] in b);
```

For the ease of examples we will consider printing statements from now on. All the SQL selection statements from section 2.1 are supported. We will limit our examples to the most important ones. The examples follow the same numbering as in the section before.

- 5 Now selecting attribute from a subquery with a where condition :

  > pr sqlexp "select a, b from (select a,b,c from d where (e=1 and f='a') or g=2);";
  ```
  pr [a,b] in ([a,b,c] where (e=1 and f="a") or g=2 in
  d);
  ```

2.2.2 Joins

Inner Joins and Difference Join

Now we can get to more complex selects implying joins :

- 14 Now select with a join and a condition on several attributes :

  > pr sqlexp "select b from c, d where c.a = d.a and e=1 and f='a';";
  ```
  pr [b] where e=1 and f="a" in (c[a:ijoin:a]d);
  ```

Left Outer Join, Right Outer Join and Full Outer Join

We can then get to more subtle joins :

- 20 Now select with the 3 different joins on several attributes :

  > pr sqlexp "select b from c, d where c.a = d.a (+) and c.e = d.e and c.f (+) = d.g;";
  ```
  pr [b] in (c[a:ljoin:a]d) ijoin (c[e:ijoin:e]d) ijoin
  (c[f:rjoin:g]d);
  ```

Joins with unqualified attributes or none

It is possible to use joins using the unqualified names of the attributes. The system will be able to differentiate them. For example if we consider a relation c with the attributes a,b and f and a relation d with the attribute e we will have the following:

```
25 > pr sqlexp "select b into destRel from c, d where c.a = d.e;";
pr [b] in (c[a:ijoin:e]d);
```

With the same output as :

```
26 > pr sqlexp "select b into destRel from c, d where a = e;";
pr [b] in (c[a:ijoin:e]d);
```

2.2.3 Aggregations

Aggregations are an important way to summarize data. They can be used the following way :

- 36 Now a more complex sum involving an alias, a group by and a having condition :

  ```
  > pr sqlexp "select sum(salary) S from employees group by
  dept having S > 10;";
  let S be equiv + of salary by dept;
  pr [S] where S > 10 in salary;
  ```

2.2.4 Multiple statements

It is also possible to write several statements at once :

```
> pr sqlexp "delete from b where a = 1;
select b from c, d where c.a = d.a and e=1 and f='a';
select b from c, d where c.a = d.a or c.e = d.e (+) and c.f = d.g
(+);
update a set b = 10 where c = 2;";
```

```
update b delete where a = 1 in b;
update a change b <- if c = 2 then 10 else b;
pr [b] in (c[a:ijoin:a]d) ujoin (c[e,f:ljoin:e,g]d);
```

The first select will be ignored because only the last one will be returned.

2.3 FILE Keyword

In both previous sections the SQL statements were directly written as the string parameter of the command. It is possible as with the *input* keyword in JRelix to specify a location on the disk to a file containing several statements to execute. The extra parameter using the file keyword allows the translator to know that the string transferred by the interpreter is not directly the SQL to translate but the path to the file where the SQL statements are. So the file is read from disk and from there the mechanism is the same as before for both SQLEXP and SQLSTMT. let consider a file named '*example.sql*' containing the following text in the current directory:

select a into destRel from b;

We will have for the following run using the the SQLSTMT command with the FILE extra parameter :

1 > sqlstmt file "example.sql";

The output corresponding :

```
destRel<- [a] in b;
```

And if we consider a file named '*example2.sql*' containing the following text in the current directory:

select a from b;

We will have for the following run using the the SQLEXP keyword with the FILE extra parameter :

2 > pr sqlexp file "example2.sql";

The output corresponding :

```
pr [a] in b;
```

Chapter 3

Implementation details

3.1 SQLSTMT Command

3.1.1 Selections

Selects

Let's consider the first of the series of runs in the previous chapter to have an idea of what is occurring in the implementation :

1 > sqlstmt "select a into destRel from b;";

The output will be :

```
destRel<- [a] in b;
```

To achieve this output we can succinctly describe what is occurring in the following way :

- First of all the input stream is redirected from the String transmitted to the translator by the JRelix system, then the lexer and the parser are invoked. The rules of the grammar are processed, the tokens recognized and finally we obtain the Abstract Syntax Tree (AST) corresponding to our SQL statement.
- Then we visit recursively each of the subtrees corresponding to a statement and store the result in a String before returning it to the JRelix system for further processing.

To be more detailed for this run we have the following execution :

- 1 > sqlstmt "select a into destRel from b;";
  `destRel<- [a] in b;`
  Algorithm :

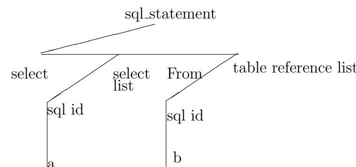

Figure 3.1: AST *select a into destRel from b;*

(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes[a]
(3) : Recognize *from* and check presence of *where*
(4) : Include relations b
(5) : Add target relation name [destRel] to the final translated statement
(6) : End of statement

In the step 1 above the target relation name is removed from the SQL to translate, stored aside and gets pasted only at step 6. This is because the *into* keyword is absent from our current grammar. Nevertheless we managed to find a workaround to be able to use it in this case (what we have not been able to do in a reasonable timeframe for the other omissions). So we intercept the SQL to translate before sending it to the parser to retrieve the target relation name, remove the into clause from the SQL and paste at the end to the translated SQL the target relation name.

Using the same logic we have for the following runs :

- 2 > sqlstmt "select b.a into destRel from b;";
  `destRel<- [a] in b;`
  Algorithm :
  (1) : Recognize *into* and store target relation name [destRel]
  (2) : Recognize qualified name [b.a] and erase the relational part [b.]
  (3) : Recognize *select* and include attributes[a]

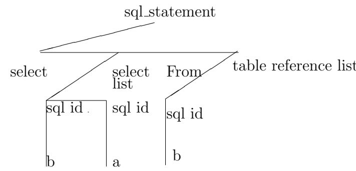

Figure 3.2: AST *select b.a into destRel from b;*

(4) : Recognize *from* and check presence of *where*
(5) : Include relations b
(6) : Add target relation name [destRel] to the final translated statement
(7) : End of statement

- 5 > sqlstmt "select a, b into destRel from (select a,b,c from d (where e=1 and f='a') or g=2);";
  `destRel<- [a,b] in ([a,b,c] where (e=1 and f="a") or g=2 in d);`
  Algorithm :

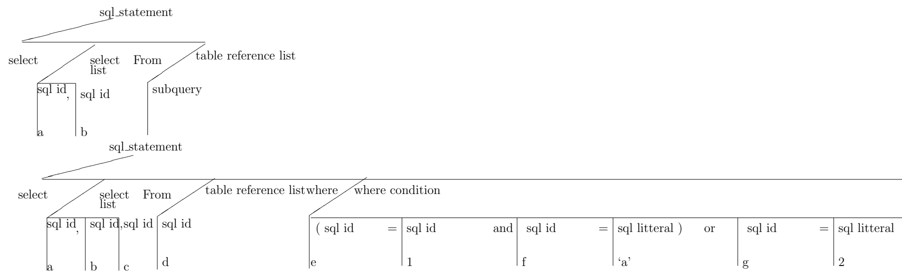

Figure 3.3: AST *select a, b into destRel from (select a,b,c from d where (e=1 and f='a') or g=2);*

(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [a,b]
(3) : Recognize *from* and check presence of *where*
(4) : Recognize subquery
(5) : Recognize *select* and include attributes [a,b,c]

(6) : Recognize *from* and check presence of *where*
(7) : Process *where* before *from* and check presence of join
(8) : Include conditions (e=1 and f='a' or g=2)
(9) : Include relations d
(10) : End of subquery
(11) : Add target relation name [destRel] to the final translated statement
(12) : End of statement

- 6 > sqlstmt "select a + b into destRel from c;";

```
let No_Name be a+b;
destRel<- [No_Name] in c;
```

Algorithm :

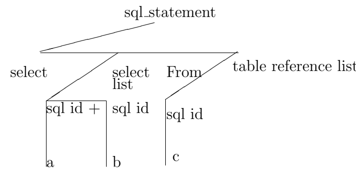

Figure 3.4: AST *select a + b into destRel from c;*

(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select*
(3) : Recognize expression [a + b] without an alias
(4) : Recognize *from* and check presence of *where*
(5) : Include relations c
(6) : Add domain algebra corresponding to the expression with the default alias
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 7 > sqlstmt "select a as b into destRel from c;";

```
let b be a;
destRel<- [b] in c;
```

Algorithm :

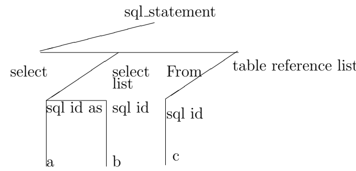

Figure 3.5: AST *select a as b into destRel from c;*

(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select*
(3) : Recognize aliasing [a as b]
(4) : Recognize *from* and check presence of *where*
(5) : Include relations c
(6) : Add renaming for the alias
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 9 > sqlstmt "select a into destRel from b where c like '%OMPUTE%';";
`destRel<- [a] where att = "c" in (grep(att) "ompute" in b);`
Algorithm :

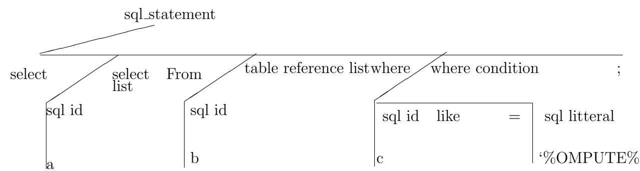

Figure 3.6: AST *select a into destRel from b where c like '%OMPUTE%';*

(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [a]
(3) : Recognize *from* and check presence of *where*
(4) : Process *where* before *from* and check presence of join
(2) : Recognize *like* condition (c like '%OMPUTE%') and generate corresponding *grep* condition
(6) : Include grep condition

(7) : Include relations b
(8) : Add target relation name [destRel] to the final translated statement
(9) : End of statement

- 10 > sqlstmt "select a into destRel from b where c + d = 1;";
`destRel<- [a] where c+d=1 in b;`
Algorithm :

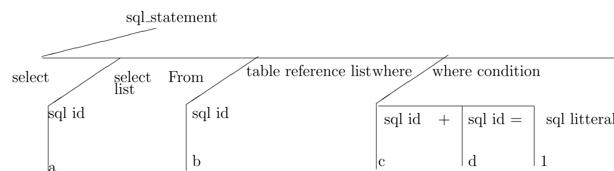

Figure 3.7: AST *select a into destRel from b where c + d = 1;*

(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [a,b,c]
(3) : Recognize *from* and check presence of *where*
(4) : Process *where* before *from* and check presence of join
(5) : Include conditions (c + d = 1)
(6) : Include relations b
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

3.1.2 Joins

IJoins

Let's continue with a runs involving a join :

- 11 > sqlstmt "select b into destRel from c, d where c.a = d.a;";
`destRel<- [b] in (c[a:ijoin:a]d);`

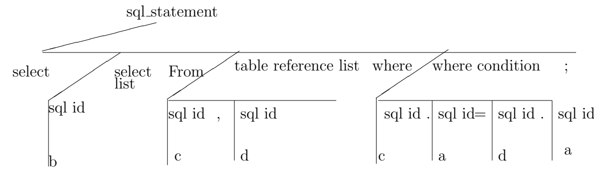

Figure 3.8: AST *select b into destRel from c, d where c.a = d.a;*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*
(Discarded) : Discard *from* processing because of the join
(5) : Process *where* before *from* and check presence of join
(6) : Process join and determine type of join on [a] attribute
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

To detect the presence of a join (step 5 above) we first try to detect a selection in the *where* and if it is not a selection then we have a join. To do that we make it in 5 steps:

- 1 First we count the number of relations involved in the *from* clause. If we have only one relation no need to go any further. There will be no join. This is done only once.
- 2 Then we break down the *where* clause to each single condition.
- 3 For each of these condition we look for the presence of a *literal* which means that it is a selection because joins involve only identifiers.
- 4 Finally on each of these conditions we test to see if the condition does not involve the same relation in which case it will again be a selection.
- 5 If any of these steps turns out to prove that it is not a selection then we have a join.

- 6 If we have a join we determine the type of the join and if we can compress it with the joins already processed, e.g. join on several attributes. We keep looping from 3 to 6 until the *where* clause is totally checked.

The following runs follow the same logic :

- 12 > sqlstmt "select b into destRel from c, d where c.a = d.a and c.e = d.e and c.f = d.g;";
`destRel<- [b] in (c[a,e,f:ijoin:a,e,g]d);`

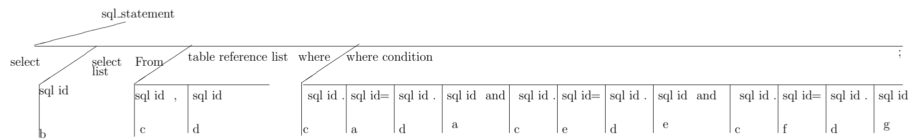

Figure 3.9: AST *select b into destRel from c, d where c.a = d.a and c.e = d.e and c.f = d.g;*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*
(Discarded) : Discard *from* processing because of the join
(5) : Process *where* before *from* and check presence of join
(6) : Process join and determine type of join on [a, e, f, g] attributes
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

LJoin, RJoin and UJoin

- 15 > sqlstmt "select b into destRel from c, d where c.a (+) = d.a;";
`destRel<- [b] in (c[a:rjoin:a]d);`

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]

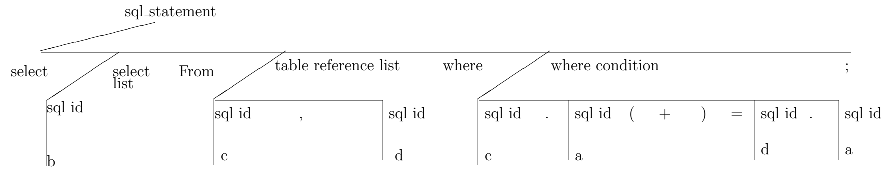

Figure 3.10: AST *select b into destRel from c, d where c.a (+) = d.a;*

(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*
(Discarded) : Discard *from* processing because of the join
(5) : Process *where* before *from* and check presence of join
(6) : Process join and determine type of join on [a] attribute
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 20 > sqlstmt "select b into destRel from c, d where c.a = d.a and c.e (+) = d.e and c.f = d.g (+);";
  `destRel<- [b] in (c[a:ijoin:a]d) ijoin (c[e:rjoin:e]d) ijoin (c[f:ljoin:g]d);`

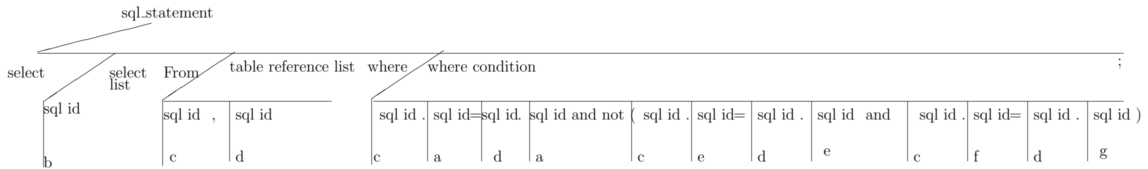

Figure 3.11: AST *select b into destRel from c, d where c.a = d.a and c.e (+) = d.e and c.f = d.g (+);*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*

(Discarded) : Discard *from* processing because of the join
(5) : Process *where* before *from* and check presence of join
(6) : Process joins and determine type of different joins on [a, e, f, g] attributes
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

Joins with unqualified attributes or none

It is possible to use joins using the unqualified names of the attributes. The system will be able to differentiate them. In order to achieve that we have to retrieve a reference to the RD relation currently in RAM for JRelix system any time a call to our translator is made. With that reference we will be able to disambiguate each attribute with an unqualified name.

For example let's consider a relation c with the attributes a,b and f and a relation d with the attribute e and let's make the following runs:

- 23 > sqlstmt "select b into destRel from c, d where c.a = c.f;";
  `destRel<- [b] where a=f in (c ujoin d);`

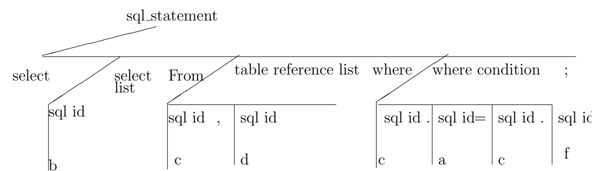

Figure 3.12: AST *select b into destRel from c, d where c.a = c.f;*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*
(Discarded) : Discard *from* processing because of the join
(5) : Process *where* before *from* and check presence of join
(6) : Include conditions (a=f)

(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 24 > sqlstmt "select b into destRel from c, d where a = f;";
`destRel<- [b] where a=f in (c ujoin d);`

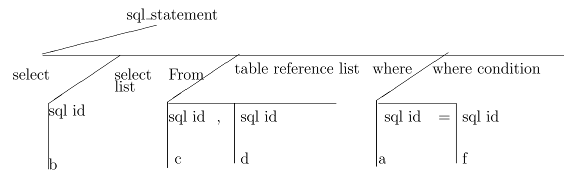

Figure 3.13: AST *select b into destRel from c, d where a = f;*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*
(Discarded) : Discard *from* processing because of the join
(5) : Process *where* before *from* and check presence of join using the RD relation
(6) : Include conditions (a=f)
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 25 > sqlstmt "select b into destRel from c, d where c.a = d.e;";
`destRel<- [b] in (c[a:ijoin:e]d);`

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*
(Discarded) : Discard *from* processing because of the join

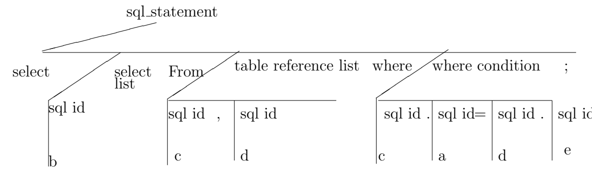

Figure 3.14: AST *select b into destRel from c, d where c.a = d.e;*

(5) : Process *where* before *from* and check presence of join
(6) : Process join and determine type of join on [a] attribute
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 26 > sqlstmt "select b into destRel from c, d where a = e;";
`destRel<- [b] in (c[a:ijoin:e]d);`

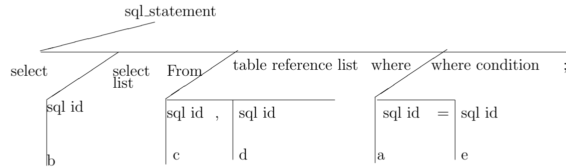

Figure 3.15: AST *select b into destRel from c, d where a = e;*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*
(Discarded) : Discard *from* processing because of the join
(5) : Process *where* before *from* and check presence of join using the RD relation
(6) : Process join and determine type of join on [a] attribute
(7) : Add target relation name [destRel] to the final translated statement

(8) : End of statement

- 27 > sqlstmt "select b into destRel from c, d;";
  `destRel<- [b] in (c ujoin d);`

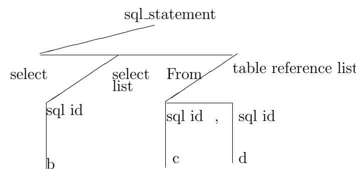

Figure 3.16: AST *select b into destRel from c, d;*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes [b]
(3) : Recognize *from* and check presence of *where*
(4) : Process *from* and determine the cartesian product
(5) : Add target relation name [destRel] to the final translated statement
(6) : End of statement

3.1.3 Aggregations

- 28 > sqlstmt "select sum(salary) into destRel from employees;";

```
let No_Name be red + of salary;
destRel<- [No_Name] in employees);
```

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and recognize *aggregation function*
(3) : Check presence of *alias* and *group by*
(4) : Process *aggregation function* and include attributes [salary]
(5) : Recognize *from* and check presence of *where*
(6) : Process *from* and include relation employees

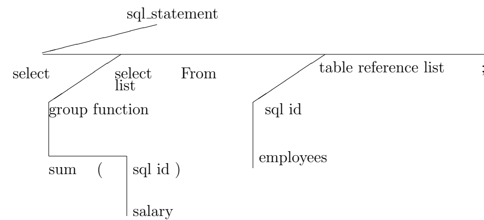

Figure 3.17: AST *select sum(salary) into destRel from employees;*

(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 29 > sqlstmt "select sum(distinct salary) into destRel from employees;";

```
let No_Name be red max of (fun + of salary order salary);
destRel<- [No_Name] in employees);
```

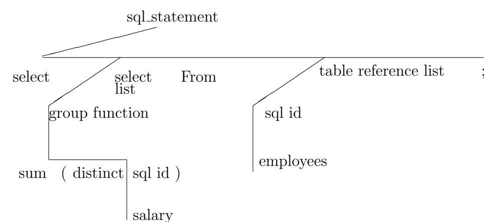

Figure 3.18: AST *select sum(distinct salary) into destRel from employees;*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and recognize *aggregation function*
(3) : Check presence of *alias* and *group by*
(4) : Process *aggregation function* and include attributes [salary]
(5) : Recognize *from* and check presence of *where*
(6) : Process *from* and include relation employees

(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 33 > sqlstmt "select count(*) into destRel from employees;";

```
let No_Name be red + of 1;
destRel<- [No_Name] in employees);
```

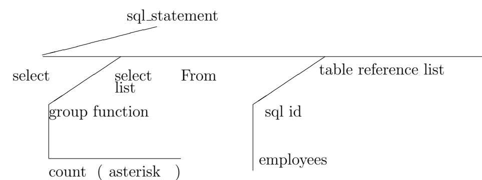

Figure 3.19: AST *select count(*) into destRel from employees;*

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and recognize *aggregation function*
(3) : Check presence of *alias* and *group by*
(4) : Process *aggregation function*
(5) : Recognize *from* and check presence of *where*
(6) : Process *from* and include relation employees
(7) : Add target relation name [destRel] to the final translated statement
(8) : End of statement

- 36 > sqlstmt "select sum(salary) S into destRel from employees group by dept having S > 10;";

```
let S be equiv + of salary by dept;
destRel<- [S] where S > 10 in salary;
```

Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and recognize *aggregation function*

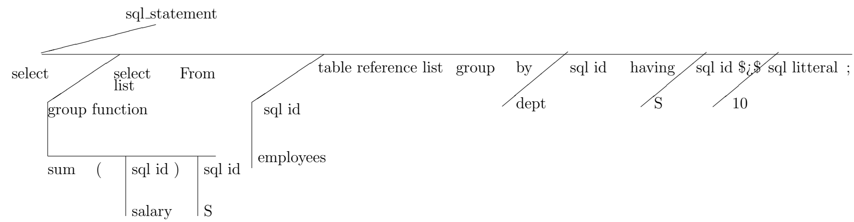

Figure 3.20: AST *select sum(salary) S into destRel from employees group by dept having S > 10;*

(3) : Check presence of *alias* and *group by*
(4) : Process *aggregation function* and include attributes [salary]
(5) : Recognize *from* and check presence of *where*
(6) : Process *from* and include relation employees
(7) : Process *having* and include conditions
(8) : Add target relation name [destRel] to the final translated statement
(9) : End of statement

3.1.4 Delete

- 37 > sqlstmt "delete from b;";
`update b delete b;`

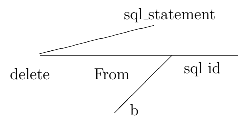

Figure 3.21: AST *delete from b;*

Algorithm :
(1) : Recognize *delete* and include relation b
(2) : Process *delete*
(3) : End of statement

- 38 > sqlstmt "delete from b where a = 1;";
  `update b delete where a = 1 in b;`

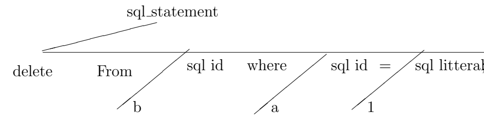

Figure 3.22: AST *delete from b where a = 1;*

Algorithm :
(1) : Recognize *delete* and include relation b
(2) : Recognize *where* and the condition
(3) : Process *delete*
(4) : End of statement

3.1.5 Update

- 39 > sqlstmt "update a set b = 10 where c = 2;";
  `update a change b <- if c = 2 then 10 else b end;`

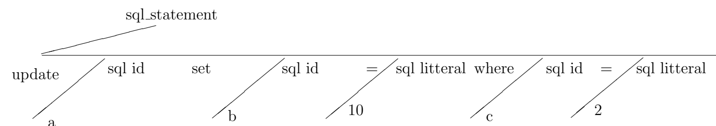

Figure 3.23: AST *update a set b = 10 where c = 2;*

Algorithm :
(1) : Recognize *update* and include relation a
(2) : Recognize *set* and include attribute [b]
(3) : Recognize *where* and the condition
(4) : Process *update*
(5) : End of statement

3.1.6 Insert

The insert command has not been added to the grammar we are using. It then can not be parsed and so it is not supported in the current version of the software. Nevertheless we can easily write the translator.

- 40 > sqlstmt "insert into employees values('Gerard', 'Tremblay', 30, 'Analyst');";
  `update employees add {("Gerard", "Tremblay", 30, "Analyst")};`

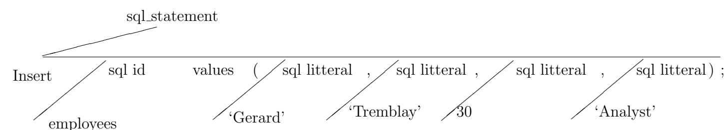

Figure 3.24: AST *insert into employees values('Gerard', 'Tremblay', 30, 'Analyst');*

Algorithm :
(1) : Recognize *insert* and include relation employees
(2) : Recognize *values* and the data to input
(3) : Process *values*
(4) : End of statement

3.1.7 General algorithm

Let's consider these 5 algorithms cited in the section above :
Algorithm :
(1) : Recognize *into* and store target relation name [destRel]
(2) : Recognize *select* and include attributes[a]
(3) : Recognize *from* and check presence of *where*
(4) : Include relations b
(5) : Add target relation name [destRel] to the final translated statement
(6) : End of statement

Algorithm :
(1') : Recognize *into* and store target relation name [destRel]

(2') : Recognize *select* and include attributes [b]
(3') : Recognize *from* and check presence of *where*
(4' Discarded) : Discard *from* processing because of the join
(5') : Process *where* before *from* and check presence of join
(6') : Process join and determine type of join on [a] attribute
(7') : Include conditions (e=1 and f='a')
(8') : Add target relation name [destRel] to the final translated statement
(9') : End of statement

Algorithm :
(1") : Recognize *into* and store target relation name [destRel]
(2") : Recognize *select* and recognize *aggregation function*
(3") : Check presence of *alias* and *group by*
(4") : Process *aggregation function* and include attributes [salary]
(5") : Recognize *from* and check presence of *where*
(6") : Process *from* and include relation employees
(7") : Process *having* and include conditions
(8") : Add target relation name [destRel] to the final translated statement
(9") : End of statement

Algorithm :
(1"') : Recognize *update* and include relation a
(2"') : Recognize *set* and include attribute [b]
(3"') : Recognize *where* and the condition
(4"') : Process *update*
(5"') : End of statement

Algorithm :
(1"") : Recognize *delete* and include relation b
(2"") : Recognize *where* and the condition
(3"") : Process *delete*
(4"") : End of statement

They will be mapped in the following general algorithm :

- The JRelix Runtime recognize the SQLSTMT command which has been added to its grammar. This addition has been done modifying the Parser.jjt to add SQLSTMT as a new token, <INSQL>, in the

Commands Tokens section and modifying the Command() Method so that it awaits the <INSQL> token with its <STRING> parameter and its optional <FILE> [1] parameter. Depending on the presence or not of the <FILE> parameter the OP_INSQL or the OP_INSQLFILE command will be transmitted to the interpreter with the <STRING> parameter.

- In the interpreter the executeCommand() Method has been modified: two cases have been added to expect the OP_INSQL or the OP_INSQLFILE command. The Translator, which consist of one class, Main, including all its elements, is invoked through its translate() Method supplying it with the String parameter, whether to consider it as a path or not, a reference to the RD relation in RAM.
- In the translator itself if the String parameter is not a filename path[2] it is divided into statements using the separator ";". (1, 1', 1") At the same time the *into* Clause is extracted and removed from Selection statements and the target relation is stored in the *intoDest* array.
- Still in the translate Method the statements are then concatenated into an InputStream which is supplied to the ANTLR parser [3] which will parse it and generate the ASTs corresponding to each statement.
- The visit() Method, which is the core method of the Translator, is then recursively invoked on each AST to produce the translated statement. This method generally translates sequentially unless some information is needed at the beginning, e.g. *group by* Clause.
- (2, 2', 2") First of all the visit() recognizes the *select* and includes the corresponding attributes.

  In the case of expression, determined using the isExpression() method, it adds them using the AddExpression() method.

  (3") In the case of aliases, determined using the isAlias() method, it adds them using the AddAliases() method.

  (4") In the case of aggregate function it processes the aggregate function using the *group by* in the visitGroup() Method.

[1] See Section 3.3
[2] See Section 3.3
[3] See Section 3.6

(1"', 2"', 3"', 4"', 1"", 2"", 3"") If it recognizes *update* or *delete* rather than *select* the whole statement is processed including the relation, the condition and the column to update in the case of an *update*.

- (3, 3', 5") In the case of a *select* statement it then recognizes the *from* and checks the presence of a *where*.
- (4, 6") In the case of a *where* it delays the *from* processing because of the join presence possibility and processes the *where* using the further parsed where condition, through our Boolean Parser [4] with the parseWhereCondition() Method before the *from* and checks the presence of a join with the isJoinGlobal() Method. If there is no join it processes the *from*.
- (4', 5', 6', 7') In the case of a join it processes the join and determine the type of the join on the selected attributes using the visitWhereJoin() Method that also includes the conditions of the statement if necessary.

  When unspecified the join attributes scan through the RD relation to find to which relation they belong to disambiguate the join. In the case of the stand alone translator there is no RD reference to disambiguate so all joins must be completely specified.
- (7")In the case of a *where* without a join or a *having* it includes conditions using the visitWhere() Method.
- (5, 8', 8")It then adds the target relation name to the final translated statement using the intoDest array.
- (6, 9', 9", 5"', 4"") The final translated statement is returned to the JRelix Runtime
- Once the translator returns the SQL translated into Aldat to the system, it is then pushed onto the Parser Stack to be executed in the next loop of the interpreter.Then the control is given back to the interpreter that can loop for the next statement to execute which is the one we just put in the Stack.

[4]See Section 3.6.4

3.2 SQLEXP Keyword

We added the SQLEXP keyword to be able to use SQL relational expression in enclosing Aldat statements. Any Aldat statement using relational expression can use it. The reason we could not use the SQLSTMT command is because as other Aldat commands it is supposed to end by an action and return to the system whereas in the case of relational expression we want to return to the system the corresponding Selection. For the ease of examples we will mostly consider printing statements.

3.2.1 Selections

Selects

Let's consider the first of the series of runs in the previous chapter to have an idea of what is occurring in the implementation :

1 > d<- c ijoin sqlexp "select a from b;";

The output will be :

```
d<- c ijoin ([a] in b);
```

To achieve this output we can succinctly describe what is occurring in the following way :

- First of all the input stream is redirected from the String transmitted to the translator by the JRelix system, then the lexer and the parser are invoked. The rules of the grammar are processed, the tokens recognized and finally we obtain the Abstract Syntax Tree (AST) corresponding to our SQL statement.
- Then we visit recursively each of the subtrees corresponding to a statement and store the result in a String before returning it to the JRelix system for further processing.

To be more detailed for this run we have the following execution :

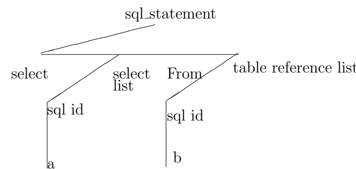

Figure 3.25: AST *select a from b;*

- 1 > d<- c ijoin sqlexp "select a from b;";
  `d<- c ijoin ([a] in b);`
  Algorithm :
  (1) : Recognize *into* absence and store system-generated intermediate relation name
  (2) : Recognize *select* and include attributes[a]
  (3) : Recognize *from* and check presence of *where*
  (4) : Include relations b
  (5) : Add system-generated intermediate relation name to the final translated statement
  (6) : End of statement

As in the previous section, it is the *into* mechanism which is also used to keep track of the intermediate relation name in the SQLEXP case. The name generated by JRelix is just transferred by the interpreter to the translator that adds it to the last Selection statement of the SQL String. That allows us to have other statements to be executed in the same SQL String while returning the result of the last Selection in the SQL translated.

Nevertheless it is worth mentioning that retrieving that correct system generated intermediate name was not straightforward. A pretty clear comprehension on JRelix internal mechanism has to be acquired first.

- 5 > pr sqlexp "select a, b from (select a,b,c from d where (e=1 and f='a') or g=2);";
  `pr [a,b] in ([a,b,c] where (e=1 and f="a") or g=2 in d);`
  Algorithm :
  (1) : Recognize *into* absence and store system-generated intermediate relation name
  (2) : Recognize *select* and include attributes [a,b]

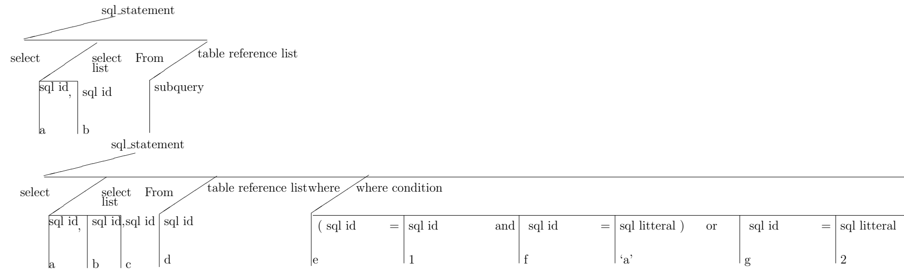

Figure 3.26: AST *select a, b from (select a,b,c from d where (e=1 and f='a') or g=2);*

(3) : Recognize *from* and check presence of *where*
(4) : Recognize subquery
(5) : Recognize *select* and include attributes [a,b,c]
(6) : Recognize *from* and check presence of *where*
(7) : Process *where* before *from* and check presence of join
(8) : Include conditions (e=1 and f='a' or g=2)
(9) : Include relations d
(10) : End of subquery
(11) : Add system-generated intermediate relation name to the final translated statement
(12) : End of statement

3.2.2 General algorithm

This general algorithm is a customized variation of the preceding one, for the SQLSTMT command, to take care of SQLEXP specificities. Only the differences are presented underneath :

- The JRelix Runtime recognize the SQLEXP command which has been added to its grammar. This addition has been done modifying the Parser.jjt to add SQLEXP as a new token, <EXPSQL>, in the *Reserved Words Tokens* section and modifying the Primary() Method so that it invokes the SQLExpression() Method which itself awaits the <EXPSQL> token with its <STRING> parameter and its op-

tional <FILE> [5] parameter. Depending on the presence or not of the <FILE> parameter the OP_EXPSQL or the OP_EXPSQLFILE command will be transmitted to the interpreter with the <STRING> parameter.

- In the interpreter the evaluateTLExpression() Method has been modified: two cases have been added to detect the OP_EXPSQL or the OP_EXPSQLFILE command. The Translator, which consist of one class, Main, including all its elements, is invoked through its translate() Method supplying it with the String parameter, whether to consider it as a path or not, a reference to the RD relation in RAM and the system-generated intermediate relation name. This system-generated intermediate relation name is provided as one of the arguments of the evaluateTLExpression() by the executeStatement() through a call to the nextTempName() Method.

- The last selection without an into is assigned the intermediate relation name in its intoDest index. All the other selections without into are ignored. From there it works exactly as having an into for that relation.

- Once the translator returns the SQL translated into Aldat to the system, it is then pushed onto the Parser Stack to be executed in the next loop of the interpreter. Then the control is given back to the interpreter that can loop for the next statement to execute which is the one we just put in the Stack.

3.3 FILE Keyword

In both previous sections the SQL statements were directly written in the *String* parameter. The extra parameter using the file keyword allows the translator to know that the string transferred by the interpreter is not directly the SQL to translate but the path to the file where the SQL statements are. Let's consider a file named *'example.sql'* containing the following text in the current directory:

select a into destRel from b;

[5] See Section 3.3

We will have for the following run using the the SQLSTMT command with the FILE extra parameter :

1 > sqlstmt file "example.sql";

The output corresponding :

`destRel<- [a] in b`;

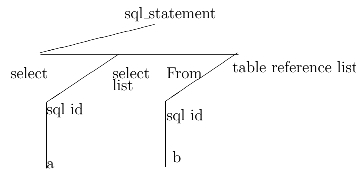

Figure 3.27: AST *select a into destRel from b;*

Algorithm :
(1) : Recognize *file* extra parameter and read from file at specified path
(2) : Recognize *into* and store target relation name [destRel]
(3) : Recognize *select* and include attributes[a]
(4) : Recognize *from* and check presence of *where*
(5) : Include relations b
(6) : Add target relation name [destRel] to the final translated statement
(7) : End of statement

And if we consider a file named *'example2.sql'* containing the following text in the current directory:

select a from b;

We will have for the following run using the the SQLEXP keyword with the FILE extra parameter :

2 > pr sqlexp file "example2.sql";

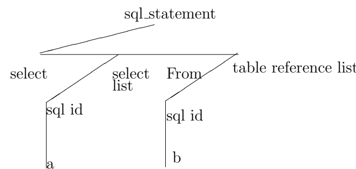

Figure 3.28: AST *select a from b;*

The output corresponding :

```
pr [a] in b;
```

Algorithm :
(1) : Recognize *file* extra parameter and read from file at specified path
(2) : Recognize *into* absence and store system-generated intermediate relation name
(3) : Recognize *select* and include attributes[a]
(4) : Recognize *from* and check presence of *where*
(5) : Include relations b
(6) : Add system-generated intermediate relation name to the final translated statement
(7) : End of statement

3.3.1 General algorithms

The general algorithms are exactly the same as the ones described in the two sections before. The only difference is rather dividing the *String* parameter into statements in the translate() method, it is used as a path to a file whose content is stored in a temporary *String* which will divided into statements. From there the behavior is exactly the same for both algorithms.

3.4 Commands grammar used which ANTLR cannot parse

Certain commands have not been added to the grammar we are using so they can not be parsed and that is the reason why they are not supported in

the current version of the software. Nevertheless we can easily support them once they would have been integrated into the grammar because structures are very close whether in Aldat or SQL. They are :

- The create command : table, view, trigger, ... etc
- The drop command
- The insert command

3.5 JRelix Configuration

Both the ANTLR and the SQL2RELIX Jar files should be included in the classpath of JRelix to use the SQL front end interface in the JRelix Runtime. The ANTLR jar is needed by the SQL2RELIX jar for the ASTs generation and navigation. And the SQL2RELIX jar is obviously needed for the SQL translation.

3.6 Tools

3.6.1 ANTLR v2.7.7 and how it operates

To generate the SQL parser we needed for our translator, we used this framework and the SQL Grammar [6] provided on their website. This allows us to think of the evolution of the parser. Through this "Meta-parser" we will be able easily to use a new grammar including new SQL standards.

So the interest was double. First it allowed us to have a "Meta-parser" that can generate the SQL parser we needed with all the AST that we could navigate through easily for our translation. Second it allowed us to plan a way to integrate the omissions in our current version of the translator from the beginning knowing that we could update the grammar as soon as we find a new one more conforming to the recent standards.

To have an idea of how the meta parser operates let's go through the steps that we did to generate our SQL Parser.

- First of all we download it from the ANTLR website[24]:

[6]See Section 3.6.3

- Then what is needed is a grammar that defines the rules of the Parser and the Lexer. In our case we directly took an SQL grammar[26] from the ANTLR website but otherwise one could have defined it. The grammar is principally composed of two parts :
  - A subclass of the generic Parser class provided by ANTLR. This subclass allows us to specify the specific rules of our SQL Parser. The rules are written using the standard Extended Backus-Naur Form.
  - A subclass of the generic Lexer class provided by ANTLR. This subclass allows us to specify the operators of our SQL Lexer. The pattern is the same as above.
- Then from that grammar we can generate our specific Parser and Lexer using the ANTLR engine. We just run it on the grammar :

  java antlr.Tool sql.g;

  It will run through the grammar and that is all that needed to be done with the Meta Parser. The files have been generated from this point on. They can now be used in the design of our translator.

  Nevertheless we will need two other classes from the ANTLR Framework :
  - ANTLR.collections.AST
    This class gives the methods to be able to navigate through the ASTs, to identify the node type and to retrieve its value. The methods that we principally use are :
    * getFirstChild()
      This methods allows us to get the first child of an AST.
    * getNextSibling()
      This methods allows us to get the next sibling of an AST.
    * getType()
      This methods allows us to get the type of a node of an AST.
    * getText()
      This methods allows us to get the the value of a node of an AST.

- ANTLR.ASTFactory
  This class is a factory class for the AST. It then gives methods to create ASTs that we used principally for the boolean parser that we developed and that is described in this chapter in a few sections.

3.6.2 Files generated

The generated files will be :

- 1 SQLLexer.java
- 2 SQLParser.Java
- 3 SQLTokenTypes.java
- 4 SQLParser.smap
- 5 SQLLexer.smap
- 6 SQLTokenTypes.txt

The files we will take a deeper look at are the Java files. The information in the other three files are redundant with the one contained in the Java Files from the point of view of our research. To use the parser easily from the main elements of the program it will be necessary to navigate through the AST generated with the parser. All this will be allowed by the Java files generated by ANTLR. Among those files, some will allow us to make the lexical analysis (1), the parsing itself (2) and the AST generation (2). We then use the AST collection class provided by the ANTLR package to easily navigate through the AST. A dictionary of the different tokens (3) , the token types, is also generated which allows us to easily identify each node of the AST and have the corresponding behavior.

Apart from the beginning where the lexing and parsing take place, we do not use the lexer and parser classes. Whereas the token types class has been used all over our translator because it allowed us to identify the clauses and to match almost one by one the clauses with Aldat. We also use a lot the AST collection class and the AST Factory class provided by the ANTLR package.

We also needed to have Java files because the current version of Aldat, JRelix, is written in Java and to ease the integration of our translator in JRelix it was much better to use the same programming language. We added the two new commands, *sqlstmt* and *sqlexp* in JRelix, to be able to directly type SQL statements in Aldat with these commands and have the output of our translator redirected to JRelix and executed. This was the main idea of our research: be able to use SQL in Aldat with the transparent SQL interface.

3.6.3 Oracle 7 SQL grammar from ANTLR website

From the ANTLR website grammar library we took this grammar [26] to generate the corresponding java files that would be used to parse the SQL, to generate the Abstract Syntax Tree and to navigate through it with our translator. This grammar is a little bit old. It is based on SQL 1 from 1986 principally. Oracle did not include ANSI statements or other evolutions of SQL until Oracle 9.

This grammar is not also totally complete. We realized this during the course of our research. It does not cover the whole scope of the complete Oracle 7. Nevertheless it still covers enough to be able to create a translator that covers the basics of SQL which was our goal.

The main reason we chose this grammar is because we wanted a SQL grammar that we can use to generate Java files. More recent grammars were available but they either did not generate Java but rather C++ files which was not what we needed or they would impose the PL/SQL Syntax which is specific to Oracle and more complex than the simple SQL syntax. So even if it was a little bit old this grammar was the most generic we could find without having to rewrite one of our own.

3.6.4 Boolean Parser

Due to the absence of analysis in the parsing of the *Where Condition* due to the grammar used in ANTLR we had to develop using the ANTLR Framework and the AST classes a boolean parser. This is principally because the AST generated for the *Where Condition* Clause is flat. At first we tried to work around it and just translate it directly. Nevertheless with the joins we finally noticed we needed to reparse it to have a real AST that we can use. To do so we used the ASTFactory class to generate the ASTs corresponding to the boolean conditions. We did not go beyond parsing the boolean

expression because we did not need to. This part would be handled by the JRelix parser the output of our translator would be fed directly to it.

The boolean parser is invoked through the parseWhereCondition() Method in the translator. It takes as argument the flat AST corresponding to the boolean expression and returns the reconstructed, layered AST corresponding to it.

Having a true AST rather than a flat one allows us to keep using our recursive approach to explore it. E.g. in the case of joins it allowed us to use the isJoinGlobal() method recursively up to the leaves of the boolean AST where the simple isJoin() method would be invoked. Whereas before creating we had to sequentially check for joins and have a very complex approach in the way we were processing *Where Condition* in particular with joins on several attributes. Generally speaking dealing with flat ASTs was much more error prone and complex than with real structured ASTs. Therefore adding this further parsing was not a luxury but a real necessity.

To give an idea of how the boolean parser operates let's consider the following boolean condition :

> e=1 and f='a' or g=2

In the context of the following select :

> select a, b from (select a,b,c from d where (e=1 and f='a') or g=2);

It will normally generate this corresponding flat AST:

And after a second parsing it will finally generate this AST :

The parser mainly recursively look for the boolean operators *and* and *or* and constructs on the way the ASTs corresponding to the boolean expressions on each side of the operator until it arrives to non boolean expression (not involving *and* or *or*) and then consider them as leaves. Our Boolean parser takes into account boolean expressions enclosed in brackets as one boolean expression and breaks it down to the non boolean expression.

In the case of our example the boolean expression is first parsed in two expressions connected with an *or* operator by calling the parseWhereCondition():

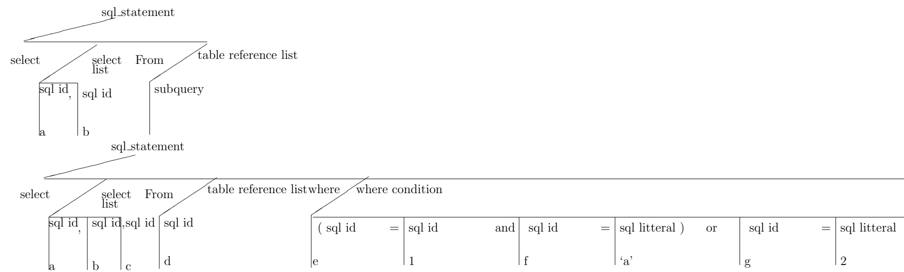

Figure 3.29: AST *select a, b from (select a,b,c from d where (e=1 and f=*`'a'`*) or g=2);*

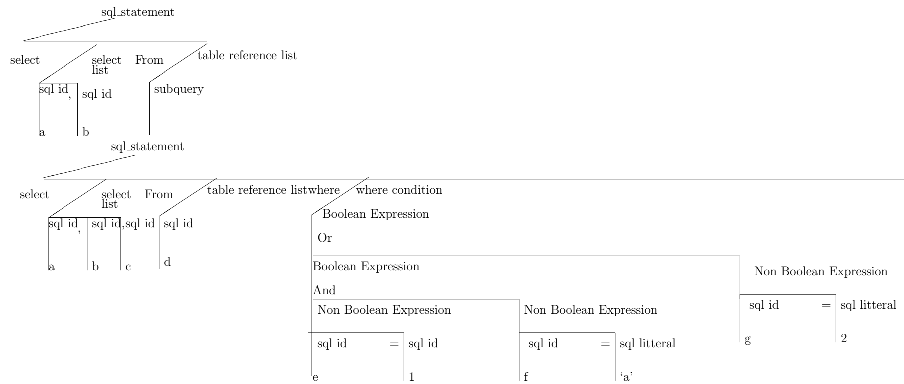

Figure 3.30: AST *select a, b from (select a,b,c from d where (e=1 and f=*`'a'`*) or g=2);*

- A boolean expression : (e=1 and f=`'a'`)
- And a non boolean expression : g=2

The first operand will be further broken down in two leaves connected by an *and* by recursively calling the parseWhereCondition():

- A non boolean expression : e=1
- A non boolean expression : f=`'a'`

Which gives the AST mentioned above.

Chapter 4

Conclusion

4.1 Recapitulation

After a brief introduction to SQL and Aldat generalities and syntaxes their resemblances and differences begin to be noticeable and some of the challenges that had to be overcome started to arises with their comparison. In particular it becomes obvious that with the vast panoply of RDBMS vendors (IBM, MySQL, SQL Server, Sybase, Oracle) and SQL versions (86, 92, 99, 2003) a choice had to be made for the SQL grammar that will be used in this research. These challenges only strengthen the motivation for this research that would use some key notions used through it (context-free grammars, LL parsers, Boolean Logic, ASTs).

After the design and the implementation of the SQL front-end for JRelix it became obvious that there will be two categories of users : the basic end-users and those interested in its internals. From there the user's manual really allows somebody who will just use the system to catch on quickly on how it operates and to see which translations he could expect from this JRelix extension. The main keywords (SQLSTMT, SQLEXP, FILE) are introduced and their behavior described through a series of examples to easily grasp their meaning. It goes from simple statements to complex multi-statements batches to translate and execute on the system.

The implementation manual then deepens into the architecture and the algorithms of the systems. Each of the keywords and how it operates on selections, updates, deletions, joins, aggregations and insertions is described in detail with the corresponding ASTs generated along the algorithm for

the same series of examples as the user's manual. The configuration of the JRelix system to incorporate the extension as well as the tools used to build it (ANTLR, the Oracle SQL grammar, our own Boolean Parser) are also described in this manual.

4.2 Future work

Several leads have been revealed by this research :

- One lead that needs to be followed would be to find a more recent grammar or complete this grammar to make it compliant with the more current SQL standard. The objective would be to make it compliant with the 2003-SQL. Three possibilities are the most plausible:
  - 1 A new grammar is posted in the ANTLR Grammar list that corresponds to the 2003-SQL. In that case the parser would be regenerated with that grammar and the backward compatibility with the current translator would need to be checked before going forward and completing the translation of the remaining grammar.
  - 2 Modify and update the current grammar to make it compliant. In this case no backward compatibilty check would need to be performed since the research would build on the current grammar and its corresponding generated parser. Nevertheless this possibily requires a certain ease with the specifications of the rules of the grammar to consistently add them.
  - 3 Modify the PL/SQL Grammar available in the ANTLR Grammar list to only keep the SQL part of the grammar. In that case also the backward compatibility needs to be checked before going forward. It is also important to make sure that with the removal of the rules intended for the PL/SQL part no new ambiguity is introduced.
- We appreciate the examiner's suggestion that we modify the rules to trigger AST-generation automatically instead of writing our own Boolean Parser. Having the grammar deal directly with all aspects of the parsing and the AST-generation without needing further programming could be less error-prone.

This has not been done this way in the first place because of a limited knowledge of the ANTLR grammar syntax at the beginning of the research.

- It is also important to support more of the SQL grammar in our translator. Our research goal was not to be extensive but to begin to show that Aldat equivalents can be written for any code in SQL, which is the de facto standard language concerning databases. To push it further it would be logical to try to think about questions as :

  – **What can Aldat do that SQL can not do?**
  One strength of Aldat is that it supports recursion by design whereas in SQL recursions are often not allowed or when they are there is always a boundary in the depth of recusion allowed (triggers, views, ...). Therefore some applications, e.g., an inference engine that derives the possible conclusions from the set of rules and the known facts, can be done easily in Aldat and are just currently impossible in SQL [13]. Since Aldat supports recursion natively with the principles of closure and abstraction, there is no limitation in the level of recursion.

  Another great limitation of SQL is that the set of aggregation functions available is limited and there can not be any functional expression that is not derived from them. Certain RDBMS gave the opportunity to write in their own procedural extension a custom-aggregated function but the syntax is complex and limited to the corresponding RDBMS if they offer it. In Aldat with the Domain Algebra there is no limitation to the functions that can be created for the aggregations. For example a simple aggregated product cannot be done easily in SQL whereas it is done in two lines for Aldat :

  ```
  > let prod be red * of possibilities;
  > pr [prod] in permutations;
  ```

  Matrix multiplication and more complex aggregations can be performed by Aldat as easily [13], in just a few lines, whereas one can imagine how complex the custom defined aggregation function may get for it. Furthermore there are no cumulative functions supplied by SQL and any of them that would be needed

would have to be defined if possible in the same schema for the custom aggregate functions. In Aldat they are already integrated in the system.

The recent support of XML in the SQL standard [23] could also be compared to the semistructured data support [8] in Aldat. This could lead to an exciting research toward the current interest in XML-related subjects.

- **Can Aldat do all that SQL can do even though Aldat is much simpler than SQL?**
  To this point we have been able to support all of the SQL capabilities that we intended to in our research, principally the more commonly used. To prove that Aldat totally subsumes SQL, this research would need to be pushed further and become extensive on the capacities of SQL.

  Nevertheless we need to keep in mind that Aldat does not provide constraint mechanisms such as the *Primary Key* to avoid duplicates because of its relational approach that does not create duplicates. In the same kind of idea, it does not provide *Indexes* mechanisms because Aldat considers indexing to be an implementation issue outside of the scope of the langage. Unlike in SQL any tuple is unique and is not identified by its row number. A tuple with a row number X can not have the exact same data as a tuple with the row number Y.

- Something else that needs to be looked at is the comparison between the queries written in SQL and their translation into Aldat to compare the syntactic complexity of these queries. For example if we look at the selections:

```
> pr sqlexp "select * from b;";
pr b;
```

It is obvious that the syntaxic complexity of the query is lower in Aldat than in SQL.

It might also be interesting to see if the same queries could have been expressed more easily if they have been directly written in Aldat. The

example that comes directly at mind is the *ijoin ijoin ijoin* example[1] which is the brute force translation from the SQL query for a join on several attributes than can collapse to the more simpler *ijoin* with the corresponding attributes. Other occurrences of these simplifications could constitute the pool of study.

[1] See Section 2.1.2

Appendix A

SQL Grammar

```
class SqlParser extends Parser;

options {
exportVocab = Sql;
k = 4;
buildAST = true;
}

tokens {
SQL_STATEMENT;
SELECT_LIST;
TABLE_REFERENCE_LIST;
WHERE_CONDITION;
SUBQUERY;
SQL_IDENTIFIER;
SQL_LITERAL;
FUNCTION;
GROUP_FUNCTION;
USER_FUNCTION;
MULTIPLY;
}

start_rule: (sql_statement)* EOF;

sql_statement: sql_command (SEMI)?
```

```
{ #sql_statement = #([SQL_STATEMENT, ”sql_statement”], #sql_statement);
}
;

sql_command:
to_modify_data
;

to_modify_data:
select_command
|| update_command
|| delete_command
;

select_command
:
select_statement ( ”union” select_statement )*
;

select_statement
:
( OPEN_PAREN select_command CLOSE_PAREN ) => OPEN_PAREN
select_command CLOSE_PAREN
|| select_expression
;

select_expression:
”select” ( ”all” || ”distinct” )? select_list
”from” table_reference_list
( ”where” where_condition )?
( connect_clause )?
( group_clause )?
( ( set_clause ) => set_clause )?
( ( order_clause ) => order_clause )?
( ( update_clause ) => update_clause )?
;
```

```
select_list:
( ( displayed_column ) => displayed_column ( COMMA dis-
played_column )*
|| ASTERISK )
{ #select_list = #([SELECT_LIST, "select_list"], #select_list); }
;

table_reference_list:
selected_table ( COMMA selected_table )*
{ #table_reference_list = #([TABLE_REFERENCE_LIST, "ta-
ble_reference_list"], #table_reference_list); }
;

where_condition:
condition
{ #where_condition = #([WHERE_CONDITION, "where_condition"],
#where_condition); }
;

displayed_column
:
( (schema_name DOT)? table_name DOT ASTERISK ) => ( (
schema_name DOT )? table_name DOT ASTERISK )
|| ( exp_simple ( alias )? )
;

schema_name
: identifier
;

table_name
: identifier
;

exp_simple : expression ;

expression
: term ( ( PLUS || MINUS ) term )*
```

;

alias
: (”as”)? identifier
;

term
: factor ((multiply || DIVIDE) factor)*
;

multiply:
ASTERISK
{ #multiply = #([MULTIPLY, ”multiply”], #multiply); }
;

factor
: factor2 (VERTBAR VERTBAR factor2)*
;

factor2
: (sql_literal) => sql_literal
|| ((PLUS || MINUS) expression) => (PLUS || MINUS) expression
|| (function (OPEN_PAREN expression (COMMA expression)* CLOSE_PAREN)) => function (OPEN_PAREN expression (COMMA expression)* CLOSE_PAREN)
{ #factor2 = #([FUNCTION, ”function”], #factor2); }
|| (group_function OPEN_PAREN (ASTERISK || ”all” || ”distinct”)? (expression)? CLOSE_PAREN) => group_function OPEN_PAREN (ASTERISK || ”all” || ”distinct”)? (expression)? CLOSE_PAREN
{ #factor2 = #([GROUP_FUNCTION, ”group_function”], #factor2); }
|| (user_defined_function (OPEN_PAREN expression (COMMA expression)* CLOSE_PAREN)) => user_defined_function (OPEN_PAREN expression (COMMA expression)* CLOSE_PAREN)
{ #factor2 = #([USER_FUNCTION, ”user_function”], #factor2);

}
|| (OPEN_PAREN expression CLOSE_PAREN) => OPEN_PAREN expression CLOSE_PAREN
|| (variable) => variable
|| expression_list
;

expression_list : OPEN_PAREN expression (COMMA expression)+ CLOSE_PAREN ;

sql_literal:
(NUMBER || QUOTED_STRING || "null")
{ #sql_literal = #([SQL_LITERAL, "sql_literal"], #sql_literal);
}
;

variable
:
(column_spec (OPEN_PAREN PLUS CLOSE_PAREN)) =>
column_spec (OPEN_PAREN PLUS CLOSE_PAREN)
|| column_spec
;

column_spec
:
((schema_name DOT)? table_name DOT)? column_name
;

user_defined_function
: ((schema_name DOT)? package_name DOT)? identifier
;

package_name : identifier ;

column_name : identifier ;

function
:

number_function
|| char_function
|| group_function
|| conversion_function
|| other_function
;

number_function
:
"abs" || "ceil" || "floor" || "mod" || "power" || "round"
|| "sign" || "sqrt" || "trunc"
;

char_function
:
"chr" || "initcap" || "lower" || "lpad" || "ltrim" || "replace"
|| "rpad" || "rtrim" || "soundex" || "substr" || "translate" || "up-
per"
|| "ascii" || "instr" || "length"
|| "concat"
;

group_function
:
"avg" || "count" || "max" || "min" || "stddev" || "sum"
|| "variance"
;

conversion_function
:
"chartorowid" || "convert" || "hextoraw" || "rawtohex" || "rowid-
tochar"
|| "to_char" || "to_date" || "to_number"
;

other_function
:
"decode" || "dump" || "greatest" || "least" || "nvl"

|| "uid" || "userenv" || "vsize"
;

pseudo_column
:
"user" || "sysdate"
;

selected_table
:
(table_spec || subquery) (alias)?
;

table_spec
:
(schema_name DOT)? table_name (AT_SIGN link_name)?
;

table_alias
:
(schema_name DOT)? table_name (AT_SIGN link_name)? (alias)?
;

link_name
: identifier
;

condition : logical_term ("or" logical_term)* ;

logical_term
: logical_factor ("and" logical_factor)*
;

logical_factor
:
(("prior") ? exp_simple comparison_op ("prior")? exp_simple) => (("prior") ? exp_simple comparison_op ("prior")?

exp_simple)
|| (exp_simple (”not”)? ”in”) => exp_simple (”not”)? ”in” exp_set
|| (exp_simple (”not”)? ”like”) => exp_simple (”not”)? ”like” expression (”escape” QUOTED_STRING)?
|| (exp_simple (”not”)? ”between”) => exp_simple (”not”)? ”between” exp_simple ”and” exp_simple
|| (exp_simple ”is” (”not”)? ”null”) => exp_simple ”is” (”not”)? ”null”
|| (quantified_factor) => quantified_factor
|| (”not” condition) => ”not” condition
|| (OPEN_PAREN condition CLOSE_PAREN)
;

quantified_factor
:
(exp_simple comparison_op (”all” || ”any”)? subquery) => exp_simple comparison_op (”all” || ”any”)? subquery
|| ((”not”)? ”exists” subquery) => (”not”)? ”exists” subquery
|| subquery
;

comparison_op
:
EQ || LT || GT || NOT_EQ || LE || GE
;

exp_set
: (exp_simple) => exp_simple
|| subquery
;

subquery
:
OPEN_PAREN select_command CLOSE_PAREN
{ #subquery = #([SUBQUERY, ”subquery”], #subquery); }
;

connect_clause
:
(”start” ”with” condition)? // the start can be before the connect by
”connect” ”by”
condition
(”start” ”with” condition)?
;

group_clause
:
”group” ”by” expression (COMMA expression)* (”having” condition)?
;
set_clause
:
((”union” ”all”) ‖ ”intersect” ‖ ”minus”) select_command
;

order_clause
:
”order” ”by” sorted_def (COMMA sorted_def)*
;

sorted_def
:
((expression) => expression ‖ (NUMBER) => NUMBER) (”asc” ‖ ”desc”)?
;

update_clause
:
”for” ”update” (”of” column_name (COMMA column_name)*)? (”nowait”)?
;

delete_command

:
”delete” (”from”)? table_alias (”where” condition)?
;

update_command
:
(subquery_update) => subquery_update
|| simple_update
;

simple_update
:
”update” table_alias
”set” column_spec EQ ((expression) => expression || subquery
)
(COMMA column_spec EQ ((expression) => expression ||
subquery))*
”where” condition
;

subquery_update
:
”update” table_alias
”set” OPEN_PAREN column_spec (COMMA column_spec)*
CLOSE_PAREN EQ subquery
”where” condition
;

identifier:
(IDENTIFIER || QUOTED_STRING || keyword)
{ #identifier = #([SQL_IDENTIFIER, ”sql_identifier”], #identi-
fier); }

;

quoted_string : QUOTED_STRING ;

match_string : QUOTED_STRING ;

keyword
:
”abs”
|| ”ascii”
|| ”ceil”
|| ”chartorowid”
|| ”chr”
|| ”concat”
|| ”convert”
|| ”count”
|| ”decode”
|| ”dump”
|| ”floor”
|| ”greatest”
|| ”hextoraw”
|| ”initcap”
|| ”instr”
|| ”intersect”
|| ”least”
|| ”length”
|| ”lower”
|| ”lpad”
|| ”ltrim”
|| ”nvl”
|| ”power”
|| ”rawtohex”
|| ”replace”
|| ”round”
|| ”rowidtochar”
|| ”rpad”
|| ”rtrim”
|| ”sign”
|| ”soundex”
|| ”sqrt”
|| ”substr”
|| ”sysdate”
|| ”to_char”

```
|| "to_date"
|| "to_number"
|| "translate"
|| "trunc"
|| "upper"
|| "user"
|| "userenv"
|| "vsize"
;

class SqlLexer extends Lexer;

options {
exportVocab = Sql;
testLiterals = false;
k = 2;
caseSensitive = false;
caseSensitiveLiterals = false;
charVocabulary = '\3' .. '\177';
}

IDENTIFIER options { testLiterals=true; }
:
'a' .. 'z' ( 'a' .. 'z' || '0' .. '9' || '_' || '$' || '#' )*
;

QUOTED_STRING
: '\'' ( ~'\'' )* '\''
;

SEMI : ';';
DOT : '.' ;
COMMA : ',' ;
ASTERISK : '*' ;
AT_SIGN : '' ;
OPEN_PAREN : '(' ;
CLOSE_PAREN : ')' ;
PLUS : '+' ;
```

```
MINUS : ’-’ ;
DIVIDE : ’/’ ;
VERTBAR : ’‖’ ;
EQ : ’=’ ;
NOT_EQ :
’<’ { _ttype = LT; }
( ( ’>’ { _ttype = NOT_EQ; } )
‖ ( ’=’ { _ttype = LE; } ) )?
‖ ”!=” ‖ ”ˆ=”
;
GT : ’>’ ( ’=’ { _ttype = GE; } )? ;
NUMBER
:
( PLUS ‖ MINUS )?
( ( N DOT N ) => N DOT N ‖ DOT N ‖ N )
( ”e” ( PLUS ‖ MINUS )? N )?
;

protected
N : ’0’ .. ’9’ ( ’0’ .. ’9’ )* ;

DOUBLE_QUOTE : ’”’ { $setType(Token.SKIP); } ;

WS : ( ’ ’
‖ ’\t’
‖ ’\r’ ’\n’ { newline(); }
‖ ’\n’ { newline(); }
‖ ’\r’ { newline(); }
)
{$setType(Token.SKIP);}
;
```

Appendix B

Sql Token Types

```
public interface SqlTokenTypes {
int EOF = 1;
int NULL_TREE_LOOKAHEAD = 3;
int SQL_STATEMENT = 4;
int SELECT_LIST = 5;
int TABLE_REFERENCE_LIST = 6;
int WHERE_CONDITION = 7;
int SUBQUERY = 8;
int SQL_IDENTIFIER = 9;
int SQL_LITERAL = 10;
int FUNCTION = 11;
int GROUP_FUNCTION = 12;
int USER_FUNCTION = 13;
int MULTIPLY = 14;
int SEMI = 15;
int LITERAL_union = 16;
int OPEN_PAREN = 17;
int CLOSE_PAREN = 18;
int LITERAL_select = 19;
int LITERAL_all = 20;
int LITERAL_distinct = 21;
int LITERAL_from = 22;
int LITERAL_where = 23;
int COMMA = 24;
int ASTERISK = 25;
```

```
int DOT = 26;
int PLUS = 27;
int MINUS = 28;
int LITERAL_as = 29;
int DIVIDE = 30;
int VERTBAR = 31;
int NUMBER = 32;
int QUOTED_STRING = 33;
int LITERAL_null = 34;
int LITERAL_abs = 35;
int LITERAL_ceil = 36;
int LITERAL_floor = 37;
int LITERAL_mod = 38;
int LITERAL_power = 39;
int LITERAL_round = 40;
int LITERAL_sign = 41;
int LITERAL_sqrt = 42;
int LITERAL_trunc = 43;
int LITERAL_chr = 44;
int LITERAL_initcap = 45;
int LITERAL_lower = 46;
int LITERAL_lpad = 47;
int LITERAL_ltrim = 48;
int LITERAL_replace = 49;
int LITERAL_rpad = 50;
int LITERAL_rtrim = 51;
int LITERAL_soundex = 52;
int LITERAL_substr = 53;
int LITERAL_translate = 54;
int LITERAL_upper = 55;
int LITERAL_ascii = 56;
int LITERAL_instr = 57;
int LITERAL_length = 58;
int LITERAL_concat = 59;
int LITERAL_avg = 60;
int LITERAL_count = 61;
int LITERAL_max = 62;
int LITERAL_min = 63;
```

int LITERAL_stddev = 64;
int LITERAL_sum = 65;
int LITERAL_variance = 66;
int LITERAL_chartorowid = 67;
int LITERAL_convert = 68;
int LITERAL_hextoraw = 69;
int LITERAL_rawtohex = 70;
int LITERAL_rowidtochar = 71;
int LITERAL_to_char = 72;
int LITERAL_to_date = 73;
int LITERAL_to_number = 74;
int LITERAL_decode = 75;
int LITERAL_dump = 76;
int LITERAL_greatest = 77;
int LITERAL_least = 78;
int LITERAL_nvl = 79;
int LITERAL_uid = 80;
int LITERAL_userenv = 81;
int LITERAL_vsize = 82;
int LITERAL_user = 83;
int LITERAL_sysdate = 84;
int AT_SIGN = 85;
int LITERAL_or = 86;
int LITERAL_and = 87;
int LITERAL_prior = 88;
int LITERAL_not = 89;
int LITERAL_in = 90;
int LITERAL_like = 91;
int LITERAL_escape = 92;
int LITERAL_between = 93;
int LITERAL_is = 94;
int LITERAL_any = 95;
int LITERAL_exists = 96;
int EQ = 97;
int LT = 98;
int GT = 99;
int NOT_EQ = 100;
int LE = 101;

```
int GE = 102;
int LITERAL_start = 103;
int LITERAL_with = 104;
int LITERAL_connect = 105;
int LITERAL_by = 106;
int LITERAL_group = 107;
int LITERAL_having = 108;
int LITERAL_intersect = 109;
int LITERAL_minus = 110;
int LITERAL_order = 111;
int LITERAL_asc = 112;
int LITERAL_desc = 113;
int LITERAL_for = 114;
int LITERAL_update = 115;
int LITERAL_of = 116;
int LITERAL_nowait = 117;
int LITERAL_delete = 118;
int LITERAL_set = 119;
int IDENTIFIER = 120;
int N = 121;
int DOUBLE_QUOTE = 122;
int WS = 123;
int ML_COMMENT = 124;
}
```

Bibliography

[1] Patrick Baker. Design and implementation of database computations in Java. Master's thesis, McGill University, 1998.

[2] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 249–264, 1974.

[3] N. Chomsky. Three models for the description of language. *Information Theory, IEEE Transactions on*, 2(3):113–124, 1956.

[4] E. F. Codd. A relational model for large shared data banks. *Communications of ACM*, 13(6):377–387, 1970.

[5] Andrew Eisenberg and Jim Melton. SQL/XML is making good progress. *ACM SIGMOD Record*, 31(2):101–108, 2002.

[6] Andrew Eisenberg and Jim Melton. Advancements in SQL/XML. *ACM SIGMOD Record*, 33(3):79–86, 2004.

[7] G. Fischer, J. Lusiardi, and J. Wolff von Gudenberg. Abstract syntax trees - and their role in model driven software development. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 38–38, 2007.

[8] Yu Gu. Basic operators for semistructured data in a relational programming language. Master's thesis, McGill University, 2005.

[9] Biao Hao. Implementation of the nested relational algebra in Java. Master's thesis, McGill University, 1998.

[10] IBM. IBM DB2 9 server SQL reference. Volume 1 & 2, September 2006.

[11] Thomas Kuhn and Olivier Thomann. Abstract syntax tree. *Eclipse Corner Articles*, 2006.

[12] T. H. Merrett. *Relational Information Systems*. Reston Publishing Co., 1984.

[13] T. H. Merrett. Aldat: A retrospective on a work in progress. *Information Systems*, 32(4):505–544, 2007.

[14] Microsoft. Microsoft SQL server 2005 books online. September 2007.

[15] Sun Microsystems. Javacc 4.0. WEBSITE : https://javacc.dev.java.net/, January 2 2006.

[16] MySQL. MySQL 5.1 reference manual. WEBSITE : http://dev.mysql.com/doc/refman/5.0/en/, December 2007.

[17] National Institute of Standards and Technology. SQL comformance test suite. WEBSITE : http://www.itl.nist.gov/div897/ctg/sql_form.htm, 1996.

[18] Oracle. Oracle 7 server SQL reference. Release 7.3, February 1996.

[19] International Standards Organization. ANSI/ISO/IEC 9075:1986 database language SQL. Technical report, ISO, 1986.

[20] International Standards Organization. ISO/IEC 8824 abstract syntax notation one. Technical report, ISO, 1990.

[21] International Standards Organization. ANSI/ISO/IEC 9075:1992 database language SQL. Technical report, ISO, 1992.

[22] International Standards Organization. ANSI/ISO/IEC 9075:1999 database language SQL. Technical report, ISO, 1999.

[23] International Standards Organization. ANSI/ISO/IEC 9075:2003 database language SQL. Technical report, ISO, 2003.

[24] Terence Parr. Antlr v2.7.7. WEBSITE : http://www.antlr.org/, November 1 2006.

[25] M. van den Brand, P.-E. Moreau, and J. Vinju. Generator of efficient strongly typed abstract syntax trees in Java. *Software, IEE Proceedings*, 152(2):170–78, 2005.

[26] Brent Wiese. Oracle 7 SQL grammar. WEBSITE: http://www.antlr.org/grammar/list, June 14 2003.

[27] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[28] Zhongxia Yuan. Java implementation of the domain algebra for nested relations. Master's thesis, McGill University, 1998.