# chiDB SQL Compiler Front-End

This is a SQL Parser and compiler frontent which can parse a reasonably large subset of SQL. The parser is generated by lex and yacc, and all other code is in C. The parser generates an abstract syntax tree (AST) representation of SQL, which is based on Relational Algebra, extended to fit closer with SQL. This extended relational algebra is termed SRA (sugared relational algebra), as it itself can be compiled into more or less pure relational algebra.

Since relational algebra primarily deals with only queries, there are separate data structures to deal with Create Table, Create Index, Insert Into, and Delete From commands (and possibly other things in the future). These don't use RA, but they do, for example, use the Expression part of the abstract syntax tree.

## Methodologies

The parser is is generated by Lex (a lexer generator) and Yacc (a parser generator). All other code is written in C, along with prototypes in Haskell.

### Lex

The lexer contains a series of regular expressions defining tokens of the language and associating them with instructions to be performed for when a given regular expression is found. The Lex utility converts a Lex file into C code which will scan an input for the next token and perform whatever instructions are to be performed when that token is found. These can be as simple as returning some integral value indicating what type of token it is (which is drawn out of an enum generated by yacc), or more complicated instructions such as dealing with comments, converting escape sequences, or storing the value of a constant expression (like an integer or string) or name of a variable.

### Yacc

Yacc is used to generate the parser. A yacc file is similar to a lex file in that it has a series of definitions of structures, and instructions for when those structures are encountered. However, in Yacc the structures are grammatical rules, and the instructions that accompany them are usually to build a parse tree (abstract syntax tree), which is a representation of the language in data structures. Yacc allows us to rapidly construct a correct and efficient parser, which gives us detailed error messages (either in parsing or in writing the parser itself), and saves us lots of time compared to a hand-written parser.

## Sugared Relational Algebra

In the ChiDB SQL parser, the instructions in the Yacc file produce a representation which we call Sugared Relational Algebra (SRA). This is an extended form of relational algebra; with several differences. For example:

* it contains as primitives multiple join types (Inner Join, Full/Left/Right outer join, Natural Join) and Intersection
* it has no rho operator, because all renaming and aliasing is contained alongside the expressions. For example in SRA, a Project structure has a list of expressions which can optionally have aliases, but in RA, a Pi structure has only expressions, and any aliases must be done with a Rho operator.
* SRA is also allowed to use \* as in SQL to stand for "all columns in the table". The step which translates from SRA to RA will expand all \*s into the actual list of columns.

The translation from SRA to RA is called desugaring. Here's an example. Say we have a table t which has columns w, x, and y:

SQL: SELECT \*, x+y as z from t;SRA:Project([\*, (Add(x, y), z)], Table(t))RA:Pi([w, x, y, z], Rho(Add(x,y), z, Pi([w, x, y, Add(x,y)], Table(t))))

A more complicated example:

SQL:select f.a as Col1, g.a as Col2 from Foo f, Foo g where Col1 != Col2;SRA:Project([(f,a,Col1), (g,a,Col2)], Select(Col1 != Col2, Join( Table(Foo,f), Table(Foo,g) ) ))Pi([Col1, Col2], Sigma(Col1 != Col2, Cross( Rho(a, Col1, RhoTable(f, Table(Foo) ) ), Rho(a, Col2, RhoTable(g, Table(Foo) ) ) ) ))

## Current Status

Currently, we have a good deal of machinery in place. The parser and lexer are finished, and all of the data structures for both SRA and RA are written, along with constructors, destructors, pretty printers, etc. There is a doubly-linked list library which is quite robust (though not fully thread-safe, but this could be reasonably easily accomplished). There is also a vector library which might be useful, for example, for string-building, or if a vector-based instead of list-based representation of columns, expressions, etc. is desired.

## Future Directions

The largest thing missing at this point is the desugarer. However, we have one written up in the Haskell language (Desugar.hs, SRA.hs, examples are in Tests.hs), and the translation of this code into C should be fairly straightforward. Haskell allows us to represent and manipulate structures of RA and SRA with ease, conciseness and correctness, and I humbly recommend any modification of the code to be prototyped in Haskell prior to coding it in C.

Finally, certain things like ORDER BY, GROUP BY, SELECT DISTINCT, sized types, etc are supported by the parser, but have little or nothing in the underlying C code to apply these. The implementation of these and other features are left to future programmers, who may wish to simply disallow them and throw errors when they are used.

## Installation and Usage

This has only been run on Mac OS X, although it should work without incident on Linux as well. Windows will presumably require installation of Lex and Yacc, along with some meddling with the makefile (if indeed make can be run on Windows), to account for different filename conventions.

Clone into the git repository:

> git clone https://github.com/thinkpad20/sql.git

Compile:

> cd sql> make

See some example tests with

> make test

Parse an arbitrary SQL file with

> bin/sql\_parser <filename>

## Condensed chiSQL grammar

Here's a summary of the SQL subset we parse in chiSQL:

sql\_queries ::= ((create\_table|insert\_into|delete\_from|select)? ';')+create\_table ::= CREATE TABLE table\_name '(' column\_dec\_list ')' column\_dec\_list ::= column\_dec (',' column\_dec)\* column\_dec ::= column\_name type ('(' INT\_LITERAL ')')? (constraint)\* | key\_dectype ::= INT | DOUBLE | CHAR | VARCHAR | TEXTconstraint ::= NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY REFERENCES table\_name ('(' column\_name ')')? | DEFAULT (literal\_value | AUTO INCREMENT) | CHECK bool\_expressionkey\_dec ::= PRIMARY KEY '(' column\_names\_list ')' | FOREIGN KEY '(' column\_name ')' REFERENCES table\_name ('(' column\_name ')')?insert\_into ::= INSERT INTO table\_name ('(' column\_name (',' column\_name)\* ')')? VALUES '(' literal\_value (',' literal\_value)\* ')'literal\_value ::= INT\_LITERAL | DOUBLE\_LITERAL | STRING\_LITERALdelete\_from ::= DELETE FROM table\_name where\_conditionselect ::= select\_statement ((UNION | INTERSECT | EXCEPT) select\_statement)\*select\_statement ::= SELECT (DISTINCT)? expression\_list FROM table (select\_constraint)\* | '(' select\_statement ')'select\_constraint ::= ON bool\_expression | USING '(' column\_names\_list ')' | WHERE bool\_expression | ORDER BY column\_name (ASC | DESC)?bool\_expression ::= bool\_term ((AND | OR) bool\_term)\*bool\_term ::= expression ('=' | '>' | '<' | GEQ | LEQ | NEQ) expression | expression IN '(' select ')' | '(' bool\_expression ')' | NOT bool\_termexpression\_list ::= expression (',' expression)\*expression ::= term (('+'|'-'|'\*'|'/') term)\*term ::= literal\_value | (table\_name '.')? (column\_name | '\*' | NULL) | '(' expression ')' | (COUNT | SUM | AVG | MIN | MAX) '(' expression ')' | '-' term column\_name ::= IDENTIFIER table\_name ::= IDENTIFIER table ::= table\_name ((AS)? IDENTIFIER)? ((',' | join) table\_name)\*join ::= (CROSS | INNER | (LEFT | RIGHT) (OUTER)? | NATURAL)? JOIN