SQL Design

**elif** self.op **==** "\*":

**return** ls **\*** self.right.value

# Document Approach

## Requirements

## Links to specs

## Discussion

## Samples

## Caveats

## Potential Changes

# Parser

The parser is the first stage of converting a SQL-92 query from string representation to the internal AST representation. The parser provides a first line of defense against malicious queries, by ensuring that queries conform to the limited subset of SQL-92 grammar that we support, and handling basic string escaping.

The parser is invoked for SQL provided by the analyst, who may be untrusted. The parser is also invoked for generated SQL that is being sent to the database, on behalf of the platform.

The parser can only enforce basic conformance to grammar. There are many additional rules that must be enforced in the AST, validator, rewriter, or private\_reader.

ANTLR offers both listener and visitor calling patterns to hook into parsing. We do not use the listener pattern. The visitor pattern can be thought of as a recursive pattern matching consumer of the parsed tree. ANTLR4 auto-generates the visitor interface in [SqlSmallVisitor.py](https://github.com/opendifferentialprivacy/smartnoise-sdk/blob/master/sdk/opendp/smartnoise/sql/parser/SqlSmallVisitor.py), and we override the methods to build the AST in [parse.py](https://github.com/opendifferentialprivacy/smartnoise-sdk/blob/master/sdk/opendp/smartnoise/sql/parse.py). This section covers just the parsing functions. The visitor functions which build the AST are covered in the next section.

## Caller API

Table

Description automatically generated

The most common calling pattern will be QueryParser().query(query\_string, metadata). This parses the supplied SQL, builds the AST, and annotates the AST with symbols from the metadata.

The queries() method will parse a batch of queries separated by semicolon.

The parse\_only() method is useful for checking conformance with the grammar, and does not use metadata or build an AST. This is particularly useful when debugging the grammar, since this method will only throw errors based on grammar problems, and will not trigger any errors related to AST construction or symbol resolution.

The parse\_expression() method may be useful for parsing fragments of text, such as arithmetic expressions.

## Grammar

The grammar, [SqlSmall.g4](https://github.com/opendifferentialprivacy/smartnoise-sdk/blob/master/sdk/opendp/smartnoise/sql/parser/SqlSmall.g4), is designed for ANTLR4. The rules are matched from bottom upwards, so the lexer rules are at the bottom of the file, and the more complex parser grammar is at the top.

The grammar starts with definition of tokens to allow the SQL-92 grammar to be case-insensitive. This is because we deal with some databases where SQL statements are lowercase, and others with strictly uppercase SQL statements.

The ESCAPED\_IDENTIFIER rule supports different database engines’ syntax for escaping identifiers, such as table names or column names that contain spaces. The QN rules are used for qualified names, such as table.column or schema.table.column.

ANTLR4 uses “Adaptive LL(\*)” parsing for ambiguity resolution. We enable strict mode when parsing, so the parser will throw errors if any ambiguities are detected. It is possible to configure ANTLR4 to raise ambiguities to the caller, where they can be resolved. However, our design is to ensure that the grammar doesn’t encounter ambiguities in supported scenarios.

When ambiguities occur in the grammar, the typical resolution is to split the ambiguous grammar into multiple components. For example, SQL-92 allows qualified column names to be used as a non-boolean expression (e.g. an integer), as in SELECT WHERE colname - 12 == 3, and also as a Boolean, as in SELECT … WHERE colname. Because the identifier matching rules are lower in the grammar file, and thus earlier in precedence, the parser would encounter ambiguity whenever a qualified name is encountered in a spot where both non-boolean and Boolean expressions are permitted. To eliminate this ambiguity, we break apart these rules into separate grammar segments. This creates some redundancy in the grammar.

The grammar syntax allows parsed values to be assigned to variable names in the parser context, via the “=” annotation, and allows rules to be named. For example:

CASE (whenExpression)+ (ELSE elseExpr=expression)? END #caseWhenExpr

Assigns the expression that comes after the “ELSE” keyword to a context variable named “expression”, and all CASE expressions which match this grammar will be processed in the “caseWhenExprVisitor” during parse.

In cases where our supported engines have different grammars, we try to support a superset. For example, TOP K and LIMIT K are supported by SQL Server and PostgreSQL, respectively. The grammar supports both. The parser can ensure that queries supply only an integer for “K”, when TOP or LIMIT are supplied, but the parser does not enforce that only one syntax is used. PrivateReader throws an error if both are supplied. This final check is something that could, conceivably, be done in the parser, and this would be the best place to catch the error. But that would complicate the grammar.

It is probably a good idea to review all similar constraints which are applied in subsequent layers, to decide if any should be moved to earlier layers. In general, we should catch errors at the earliest possible processing stage.

## Build

After editing the grammar file (SqlSmall.g4), you need to re-generate SqlSmallVisitor.py and the supporting files. This can be done by invoking make from the sql/parse folder. The default make target builds the required python files, if the SqlSmall.g4 grammar file has been touched.

You can use make gui to try out the grammar changes and see a visual parse tree. You can type a query (or query batch, with queries separated by semicolon), and then press CTRL-D on the keyboard to see what the parser thinks of the input. For example, the below is the parse tree for query:

SELECT educ, SUM(income) AS income FROM PUMS GROUP BY educ ORDER BY income DESC LIMIT 10.

Diagram

Description automatically generated

## Unit Tests

Unit tests for the parser are included in the [test\_ast.py](https://github.com/opendifferentialprivacy/smartnoise-sdk/blob/master/tests/sdk/query/test_ast.py) harness. Although this harness performs AST tests, it also performs tests for bare parsing, including queries that should pass, and queries that should fail. Test queries are stored as text batches with extension \*.sql, in the [queries](https://github.com/opendifferentialprivacy/smartnoise-sdk/tree/master/tests/sdk/query/queries) folder, organized by processing stage. For example, queries in the [queries/parse](https://github.com/opendifferentialprivacy/smartnoise-sdk/tree/master/tests/sdk/query/queries/parse) folder will be tested against only the parser, while queries in the [queries/validate](https://github.com/opendifferentialprivacy/smartnoise-sdk/tree/master/tests/sdk/query/queries/validate) folder will be tested against parsing, AST, and validation.

The test harness has a GoodQueryTester and BadQueryTester, which filter out SQL batches that are intended to pass, and queries that are intended to fail. All SQL batches named \*\_fail.sql will be tested to ensure failure of parsing, and all other batches in the parse folder will be tested for success.

Any changes to the grammar should be accompanied by success and failure unit test queries, placed in the [queries/parse](https://github.com/opendifferentialprivacy/smartnoise-sdk/tree/master/tests/sdk/query/queries/parse) folder.

# Abstract Syntax Tree

The AST is the in-memory representation of the parsed query tree. It’s goals are:

* Transform the parse tree into a Python object model that can be validated for differential privacy
* Object model useful for the rewriter
* Allow serialization back to SQL text
* Helper methods such as evaluate, find\_nodes
* Object model for propagation of data curator metadata, such as sensitivity and symbols.

## AST Construction

The AST is built in the QueryParser().query() method in [parse.py](https://github.com/opendifferentialprivacy/smartnoise-sdk/blob/master/sdk/opendp/smartnoise/sql/parse.py). The root of the parse tree is a batch, which includes one or more query objects. We use ANTLR’s visitor pattern, which walks the parse tree recursively. Each visitor has a visit() method, which extracts the parsed subtree and returns the appropriate typed objects. Thus, we start with the BatchVisitor object, and BatchVisitor invokes QueryVisitor for each query in the batch. The QueryVisitor.visit() method returns a Query object, and the BatchVisitor.visit() method returns a list of Query objects.

All objects in the typed AST object model are contained under the opendp.smartnoise.\_ast namespace.

The AST object model is closely related to the grammar, but eliminates many intermediate grammar elements which are not needed outside of parsing.

AST construction also merges grammar fragments that may have been broken out separately in the grammar to avoid ambiguities. For example, the ColumnBoolean AST object, which represents a column that is to be treated as a Boolean (e.g. in SELECT … WHERE married), is created in the BooleanExpressionVisitor’s visitQualifiedColumnName method. The grammar breaks out qualifiedColumnName into multiple fragments, to avoid ambiguities when parsing, and these differing contexts are collapsed back into the appropriate objects by the AST.

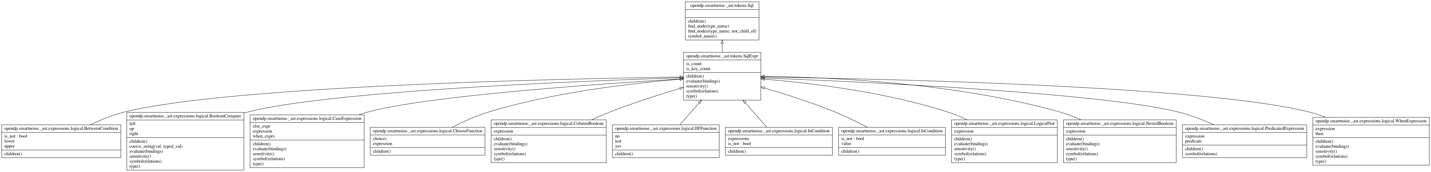
## Object Model

### AST (Query)

Diagram

Description automatically generated

### Logical



### Numeric

Diagram

Description automatically generated

## NamedExpressions

## Column

## Utility Methods

### Find\_node, find\_nodes, etc.

### Children (for recursion)

## Evaluation

## Serialization

The invisible sequence

## Walk relations

## Symbols overview (shadow AST)

## Sensitivity

# Symbol Attachment

## Shadow AST (purpose is to attach metadata)

## (what validation is done here?)

## Relations

## TableColumns (pulls in metadata)

## How name resolution is done

## What if wrong NameCompare?

# NameCompare

# SqlReader

## Execute

## Execute\_df

## Connection properties

### Host

### Password

### Port

### User

### Driver-dependent properties

## Returning cursors

# Validator

## Overview

## List of Rules

### Inner aggregates

### Presence or absence of keys

### Counts for korolova, etc.

### Supported aggregates

## Does this need NameCompare?

## What about rules that involve propagation of sensitivity? Others not obvious from AST?

# Rewriter

## Choosing what to add noise to

## Valid options (clamp\_dims)

## NameScope

### Anonymous Names

## Pushing expressions like VAR

### Should we swap DP\_SUM for diff priv SUM, etc.?

## Pushing expressions that use the core library

## Pushing expression that needs histogram or other computed columns

# PrivateReader

## Valid options (clamp\_dims)

## Row-based aggregates

## Functional (map/filter/sort)

## This duplicates some validation rules. Is there a reason why?

## Are there some validation rules here that aren’t in validator? If so, why?