SQL Dev Design

Parser

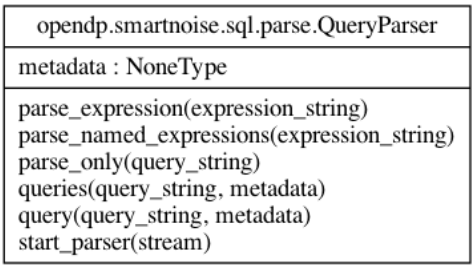
The parser is the first stage of converting a SQL-92 query from string representation to the internal AST representation. The parser provides a first line of defense against malicious queries, by ensuring that queries conform to the limited subset of SQL-92 grammar that we support, and handling basic 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, and we override the methods to build the AST in parse.py. This section covers just the parsing functions. The visitor functions which build the AST are covered in the next section.

Caller API



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, 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 suggested 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.

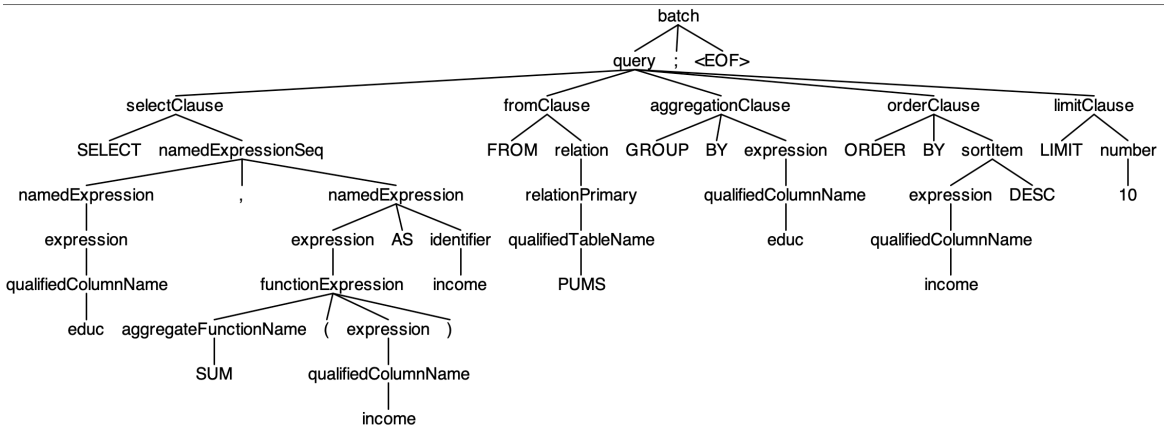
It is probably a good idea to review all similar constraints which are applied in subsequent layers, to decide if any should be moved to earlier layers. In general, we should catch errors at the earliest possible processing stage.

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.
```



Unit Tests

Unit tests for the parser are included in the 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 folder, organized by processing stage. For example, queries in the queries/parse folder will be tested against only the parser, while queries in the 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 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. 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.

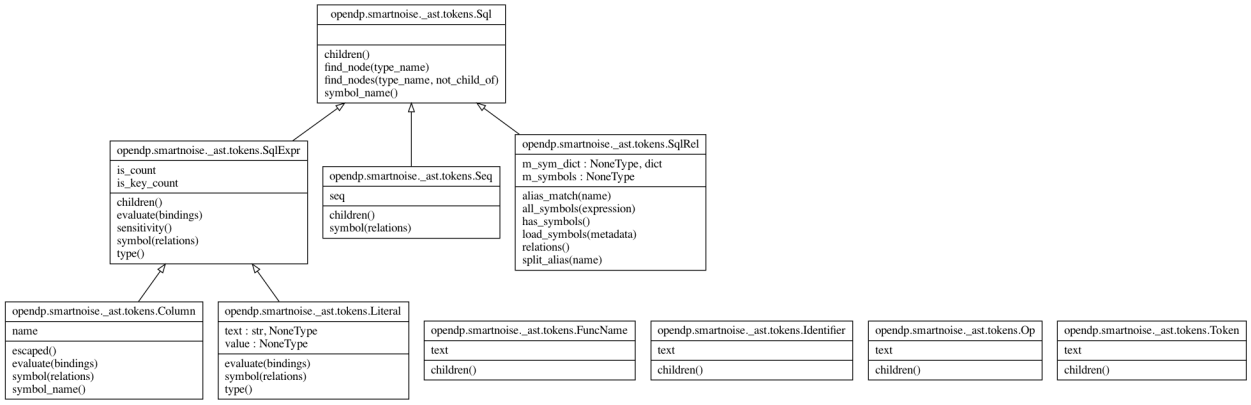
The AST object model is closely related to the grammar, but eliminates many intermediate grammar elements which are not needed outside of parsing.

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

Diagrams are shown here as PNG. EPS versions of the diagrams are available in the sqldocs repo.

Tokens



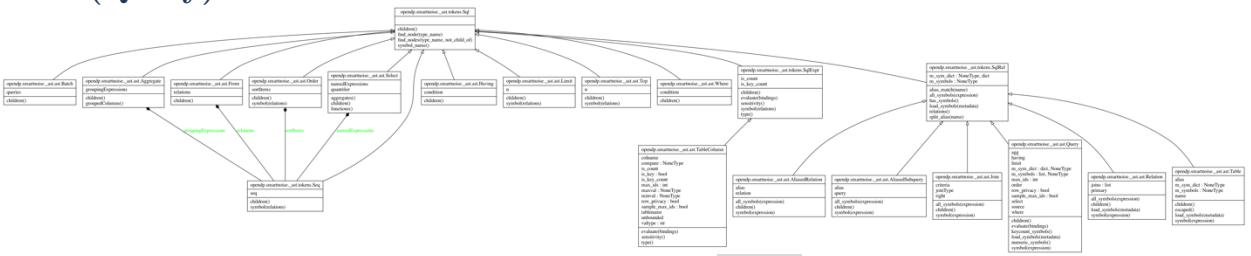
The root objects are defined in `tokens.py`. These serve as root types with some helper methods, and allow disambiguation between `Column` and `Literal` identifiers.

The `Op` class is used for operators, such as `<` or `+`.

The `Token` class is a placeholder for grammar fragments that have no special processing, but which need to round-trip when being serialized back to text. This is mainly useful for specifying tokens that need to be serialized, such as (and). The Tokens class should not be used as a destination object in AST construction, since this could result in unsafe behavior.

The `Seq` class represents sequences of other classes, such as `NamedExpressions`. We use `Seq`, rather than python lists, so that serialization can add commas automatically.

AST (Query)



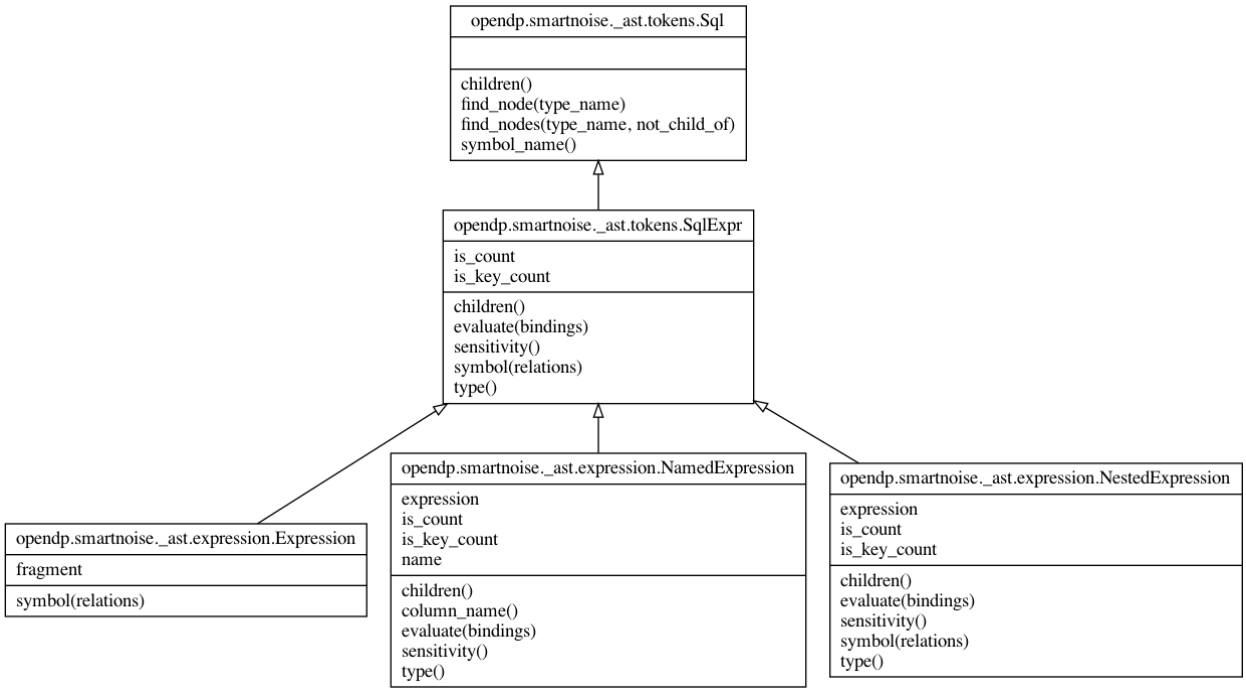
The top-level SQL grammar objects are defined in `ast.py`.

All SQL grammar objects inherit from the `Sql` base class, defined in `tokens.py`. This base class implements the `find_nodes()` helper method, discussed below.

The classes for simple SQL clauses, such as `Select`, `Having`, `Where`, `Order`, `From`, and `Aggregate`, all inherit directly from the `Sql` object. Note that the `Aggregate` object is used to represent a GROUP BY clause.
All SQL relations defined in `ast.py`, such as `Table`, `Query`, and `Join`, inherit from the `SqlRel` subclass, also defined in `tokens.py`. The `SqlRel` subclass extends `Sql` by adding methods and properties used for symbol resolution, discussed below.

The `TableColumn` class is used to represent a *resolved* column, not a parsed column reference from the grammar. The AST uses the `Column` class, defined in tokens.py, to represent a column that has been parsed, but not yet resolved. The symbol resolution process converts `Column` classes to `TableColumn` classes, by matching metadata with the query columns.

Expressions



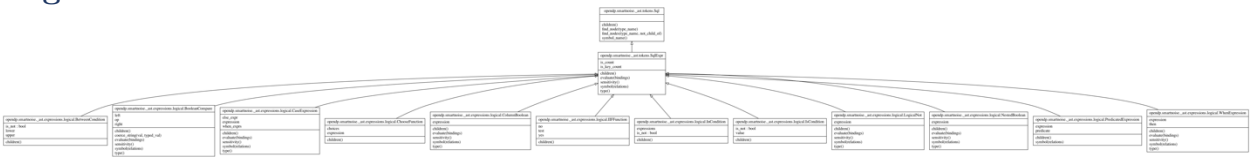
The `NamedExpression` object represents any evaluable column which may or may not have an output alias. The SQL SELECT statement is a sequence of one or more named expressions, such as `SELECT AVG(income) AS inc, SUM(2 * educ) AS educ`.

A `NestedExpression` is an anonymous expression surrounded by parenthesis.

The bare `Expression` object can be used as a placeholder for expression grammar fragments which do not have a dedicated AST object, which need to round-trip through serialization. The `symbol()` method on this class raises `ValueError` exception, because

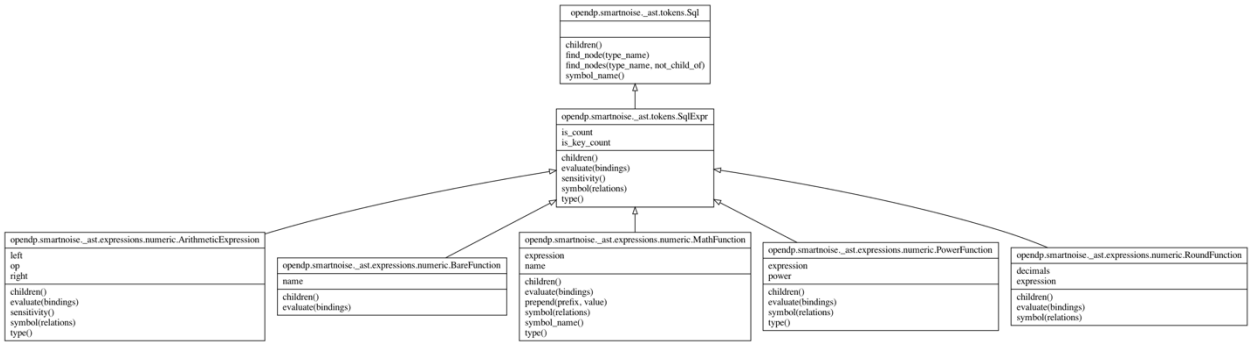
there is no way to attach symbols for arbitrary unknown grammar. This means that this class can be used in round-tripping ASTs, but cannot be used in any queries that need differential privacy.

Logical



The `expressions/logical.py` module defines a variety of classes representing SQL logical operations. These classes all inherit from the `SqlExpr` subclass, which defines the helper method, `evaluate()`.

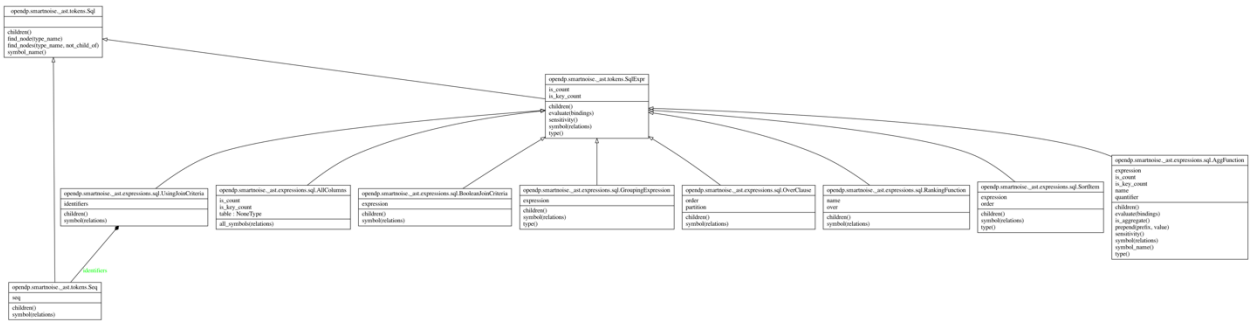
Numeric



`expressions/numeric.py` defines the numeric expressions, all inheriting from the `SqlExpr` subclass and implementing `evaluate()`.

`BareFunction` maps to `bareFunction` in the grammar, and is used to represent functions with no arguments, such as PI() or RANDOM().

Sql



A handful of additional expression types are defined in expressions/sql.py.

The `AllColumns` class is a special case, representing either `*` or `table.*`. Because it represents all columns in one or all relations, symbol resolution must walk the relevant relation or relations and bring in all columns. This step must be careful to prohibit duplicate column names.

Resolving the `AllColumns` object to the underlying column names creates some complication for serialization. Serialization operates by recursively walking and serializing all `children()`. In the case of symbol resolution, we want all of the children to be available in the AST, but they should be emitted as `*` on round-trip.

(What do we do about missing columns from metadata? Rewriter should ensure calling all columns by name)

The `AggFunction` class represents an aggregation, such as `SUM` or `AVG`. Because these functions are often called in an anonymous named expression (e.g. `SELECT SUM(income * 100) FROM foo`), SQL engines have the ability to automatically choose the output column name. The `symbol_name()` method on `AggFunction` attempts to return a sensible default column name. Because different engines have different default column naming rules for aggregate functions, this means that the AST will not always result in the same column names as a query fed directly to the engine. If this is a concern, we could push the `symbol_name()` functionality to the `SqlReader` subclasses, allowing engine-specific overrides. However, this has not been a problem so far.

Serialization

Every descendant of Sql must implement the python built-in `__str__()` method, which emits the text serialization in SQL grammar for that component.

The `children()` method must be implemented, and returns the list of children of the component, in the correct serialization order. For example, the `BareFunction` class returns three children: the function name, `Token(“(“)` and `Token(“)”)`. This allows serialization to function by concatenating the `str()` output of all children in order.

Evaluation

The `evaluate()` method is defined on all `SqlExpr` descendants. It allows a python dictionary to be passed in with variable bindings, to be plugged in anywhere that a column name is referenced. If no column names are referenced in the expression, `bindings` can be `None`.

Different database engines have different rules for identifier matching. For example PostgreSQL may have column names which are case-sensitive, and SQL Server does not. Because of this, the `evaluate()` function uses the `NameCompare` class on the engine-specific `SqlReader`. This could become a source of bugs, if a query written for one engine is executed against via an AST that uses a different engine's name comparison rules.

Helpers

The `find_nodes()` and `find_node()` helper methods can be used to recursively walk the AST to find a specified node, based on type. The function allows an option parameter specifying a node that's descendants should not be searched.
The `xpath` and `xpath_first` helper methods can be used to select nodes in the SQL AST using XPath syntax.
The `is_count` helper property should return true for expressions that should be treated as counts, and `is_key_count` returns true for expressions that count the private identifier.

Unit Tests

The unit tests are contained in test_ast.py, as described above in the Parsing section. In addition to the standard parsing tests, the test queries for the AST go through an extra set of tests, which include round-tripping from AST back to text and ensuring the query is unchanged, and round tripping back to a new AST and ensuring objects test as identical.

Symbol Resolution

Symbol resolution is the process of annotating all referenced columns in the query with the appropriate metadata. The metadata is organized in tables and columns, so the first step in symbol resolution is to walk all relations in the query, ultimately matching table names with metadata. This step is implemented in the `load_symbols(metadata)` function that is implemented on `SqlRel` subclasses which can reference tables or queries. This pass results in the `m_symbols` and `m_sym_dict` properties being populated. These data structures represent all fields that are queryable from the relation. In SQL, a relation is simply a collection of rows and columns, with columns being referenceable by name. `load_symbols()` is the way that we expose al queryable columns on any relation.

`m_symbols` is an ordered list of symbols, with order being important for the main `Query` object. `m_sym_dict` is just a dictionary with the symbols keyed for lookup by name. The

dictionary may be missing columns which are anonymous (or may contain them with an inferred name).

(We should make a call one way or another here).

`m_symbols` is a list of (name, symbol) tuples, where the name is the value used for referencing the symbol, and the symbol is a typed object or object tree. For any `Table` object, `load_symbols()` pulls in all column names from the metadata, as `TableColumn` objects.

Other types of relations are more complex, as they can include columns from multiple tables, or can even construct totally new columns, as in the case of a `Query` relation. In all cases, however, the `m_symbols` property will list the columns that are queryable from that relation.

Once the queryable columns have all been loaded into the lookup tables, with attached metadata, the second step of symbol resolution begins. This step resolves all identifiers that are used in expressions, to map them to underlying table columns. This is done via the `symbol(expression)` method, which is traversed recursively until all `Column` objects have associated `TableColumn` objects. For expressions that do not reference a `Column`, the `symbol()` function simply returns a clone of the object's tree. You can think of `symbol()` as building a "shadow" AST which has `Column` objects replaced with `TableColumn`s containing metadata.

(is this really a clone?)

(why do we return a shadow? Why not replace in-place?)

As the `symbol()` function is called recursively, it handles aliasing. The following example uses superscript annotations to show the steps that are taken in a symbol resolution that has table aliases.

```
SELECT SUM(o.Sales⁴)³ FROM Order² o, Customer¹ c USING cid
```

1. `load_symbols` attaches all of the `TableColumn` objects from `Customer` metadata to that `Table` object's `m_symbols`
2. `load_symbols` attaches all of the `TableColumn` objects from `Order` metadata to that Table object's `m_symbols`

3. The `symbol()` method gets called on `AggFunction`, which returns an `AggFunction` with expression (which is a `Column` in the original non-resolved AST) replaced by the call to the child object's `symbol()`
4. The `Column` `symbol()` walks through all relations in the query, skipping any relation where the alias doesn't match. In this example, the `Customer` relation will be skipped, and the `Sales` column will be matched to the `TableColumn` in `Order`.

As with evaluation, the mapping of `symbol()` to `TableColumn` depends on engine-specific rules for identifier matching. We use the engine-specific identifier matching code implemented in the `NameCompare` of the engine-specific `SqlReader`.

Because the job of symbol resolution is to ensure that proper sensitivity and other important constraints (such as `max_ids`) propagate through the query, bugs in this code could lead to privacy exploits. For example, if an adversary can trick a query into using a metadata column with lower sensitivity, privacy would be compromised. There are several tricky edge cases with JOINS and nested relations that could expose bugs if not carefully tested. For example, aliases can be nested and scoped. This code should be rigorously reviewed and tested before re-enabling JOINS.

We do not expose the `m_symbols` property on all `SqlRel` objects. The current design philosophy is that `m_symbols` is only exposed on objects which create columns, and those include only `Table` and `Query` objects. All other relations expose their symbols through the `all_symbols()` method, which simply recurses to find columns from their source relation(s). This might not be an optimal design, since it requires callers to know whether a relation can create columns or not, and perhaps it would be better if all relations can be treated the same. This might be an issue when/if we add support for deeply nested SELECT subqueries and JOINs.

Symbol resolution occurs before `Validator` is called, because validation relies on information obtained through symbol validation. Some level of validation happens during symbol resolution, and is not postponed to `Validator`. For example, queries must reference columns that are available in the metadata. Because symbol resolution is triggered automatically as part of parsing (as long as metadata is supplied), this means that the API limits the degree to which callers can construct ASTs which are not valid for differential privacy.

This separation is not as clean as could be imagined. Ideally, the AST would allow any number of privacy-invalid queries to be constructed, as long as the queries adhered to

the formal grammar, and implied grammar such as non-duplication of identifiers. Privacy-enforcement would only kick in at the `Validator` stage, and all stages after. This is important, because the AST is used in other parts of the platform, for privacy-important work that is nevertheless not intended to validate. For example, reservoir sampling is implemented in the rewriter using AST fragments rather than strings, for all of the other benefits that come from using the AST.

The original intention was that AST parsing would branch based on presence of metadata, such that symbol resolution would be a separate and optional step. However, this code path is not well-factored or well-tested, and the current assumption is that metadata is always passed in to parser, and symbol resolution always happens automatically at parse.

Sensitivity

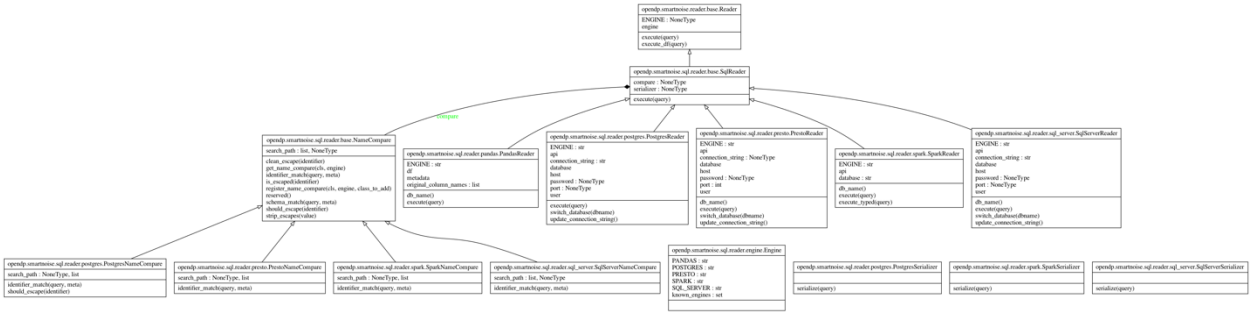
The `sensitivity()` helper method propagates the sensitivity for columns through the AST from metadata. This is how the postprocessing determines how much noise to add, so bugs here can impact privacy directly. The metadata allows columns to specific `unbounded` sensitivity, which allows the column to be used in a COUNT or GROUP BY, but not in SUM or related numeric aggregates. `sensitivity()` returns `None` when metadata are unavailable, or where sensitivity doesn't make sense (such as categorical data).

`sensitivity()` attempts to handle interval arithmetic. For example, `SUM(2 * foo)` will result in sensitivity being doubled, while `SUM(2 + foo)` will result in unchanged sensitivity.

For columns marked as unbounded in the metadata, sensitivity is set to `np.Inf`. In cases where sensitivity is not sensible, `None` is returned. Care should be taken to make sure that columns with sensitivity of `None` are not returned improperly.

Note that we currently use `add-remove` neighboring definition, and core defaults to `substitute`. For parity with core, we should update to support both, and default to `substitute`. Stochastic validator will need to support both.

SqlReader



The `SqlReader` base class represents a connection to a database that returns rowsets from SQL queries. It derives from the `Reader` base class, which returns rowsets from arbitrary (potentially non-SQL queries). Both classes implement an `execute()` method, which takes a query in text format and returns a rowset, as an iterator over tuples.

The `execute()` method should always return an iterator, and should stream rows as they are received from the database.

Both classes have an `execute_df()` helper method, which converts the output results to a Pandas DataFrame before returning.

(Add metadata validation here? Or in validate? Or in ConnectionMetadata?)

Each engine-specific reader (e.g. `SqlServerReader`, `PostgresReader`, `PandasReader`) derives from `SqlReader`. The purpose of each custom reader is to:

1. Accept connection information, and establish a connection with the database, wrapping whatever engine-specific libraries are needed to support that database.
2. Implement identifier comparison rules that are engine-specific
3. Plug in serialization rules to ensure executed queries use engine-compatible syntax
4. Wrap engine-specific idiosyncrasies in a common interface

Connection Information

Each constructor tries to copy the parameter definition and order that are common to that engine's connection API. Going forward, we will recommend callers to use `SqlReader`'s `from_connection` factory method, which takes an engine parameter and instantiates the correct concrete reader implementation, attaching the appropriate

`NameCompare` and `Serializer`. A future update will allow probing of a connection to determine the engine, making the `engine` parameter optional.

NameCompare

Implementations of the `NameCompare` base class, shown in the diagram above, override the `identifier_match` method, to support engine-specific identifier matching rules. For example, table names in Postgres are case-sensitive, while table names in SQL Server are not. Table names with spaces can be escaped with square brackets in one database engine, and quotes in another. The name compare also handles default schema search rules. For example, `table1` and `dbo.table1` refer to the same table on SQL Server, since `dbo` is the default schema.

`identifier_match` takes two parameters, `query_identifier`, and `meta_identifier`, and reports on whether or not they match. This method is called during symbol resolution and evaluation to link identifiers used in the SQL query with objects in the database that are described in the metadata.

The implementor's only responsibility is to implement the engine-specific equality rules. Callers (e.g. symbol resolution) must carefully consider edge cases when calling this method. For example, if a caller attempts to resolve symbols by stopping at the first match, this could result in privacy bugs, because duplicate columns are engine-specific.

Serializer

The `Serializer` base class, shown above, is intended to implement engine-specific rules for serialization of ASTs to string. This is envisioned as a sanitization and canonicalization step that can help ensure compatibility and mitigate SQL injection attacks. There are two places where sanitization on query serialization is done with simple string replaces. Doing sanitization through the AST (e.g. with `find_nodes`) is more robust. Example serialization benefits could include:

- Quoted strings can be converted from generic query representation to engine-specific quotes. We currently strip all quote characters, which is overly aggressive.
- Automatically handle escaping rules, like identifiers with spaces
- Handle casing conventions (e.g. Postgres all lowercase, SQL Server all upper)
- Switch engine-specific grammar. For example, TOP K and LIMIT support are disjoint by engine. Subquery syntax can be idiosyncratic.

- Canonicalize identifiers

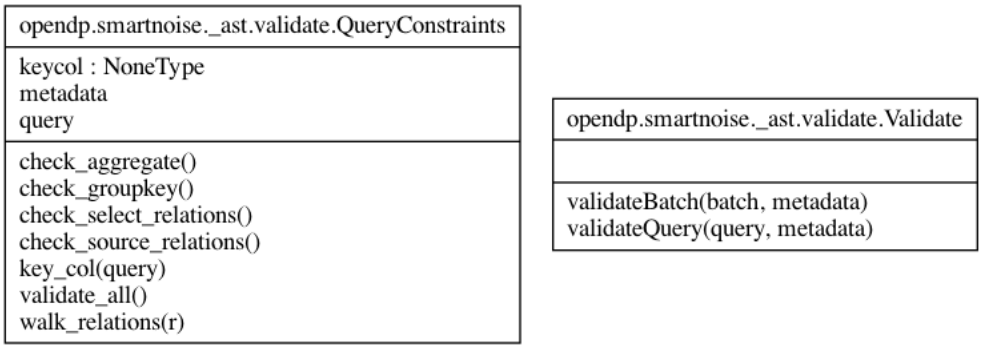
Out of Scope

Other engine-specific behavior not mentioned here, is out of scope for this design. These may be important to consider in the future. Examples include collations and code pages, date/time handling rules, and behavior of inferred column names.

Validator

Validation occurs after symbol resolution, but before the query is rewritten. The goal of validation is to ensure that the query meets all requirements to be processed with differential privacy.

The philosophy is that AST and symbol resolution may freely represent queries that are disallowed from a differential privacy standpoint, and validation should be used to verify that a given query meets the requirements to be processed with differential privacy.



The `validateQuery()` method receives a Query AST object and metadata, and checks all `QueryConstraints` objects in `_ast/validate.py` by executing each `check_()` method against the AST. Each constraint defined in the object is tested for The design is intended to allow constraints to be defined and tested independently and incrementally. Constraints can use `find_nodes` helper method to be more declarative.

The validation code was originally designed to return a tuple with True or False, and applicable error message for any failed constraint. The intention was that validation could be used to see a full list of failing constraints. However, several of the constraints

now raise exceptions, which means the validator will fail fast on error. This should be cleaned up to use only one pattern.

There are some additional validation steps which take place before the validator, during symbol resolution, and some which take place after, in the rewriter and private_reader. These should be examined, and any which can be duplicated in the validator should be duplicated. Philosophically, the validator should provide as complete a validation as possible, even if the caller never intends to execute the query with private_reader.

Some examples of validation that should be duplicated from symbol resolution or private_reader to validator include:

- Ensuring that columns in aggregates do not have unbounded sensitivity
- Ensuring that bounded columns have both upper and lower

It is fine for subsequent layers to catch the same errors, for defense in depth. It is less obvious that preceding layers should catch error that are unrelated to privacy (see discussion under symbol resolution section).

Validation should use `NameCompare` when matching identifiers.

Out of Scope

Currently, the validator does not validate some user contribution limits, because we do not support joins or subqueries where these may be problematic. However, we will need to figure out how to validate these in the future, and it's not clear that the AST+metadata will be sufficient information for the validator to reason about these queries.

These queries may arise even without joins. For example, `SELECT url, COUNT(DISTINCT userId)`.

Rewriter

The rewriter takes a valid, analyst-provided query, and rewrites it into a query plan that can be used for differential privacy. Rewriting the query plan allows us to add functionality such as:

- Clamping values to be within the upper and lower bound
- Reservoir sampling IDs to enforce user contribution

- Rewriting certain supported aggregates to execution forms. For example AVG(x) becomes SUM(x) / COUNT(x)
- Obtaining hidden columns used for filtering or computation. This step allows re-use of noisy answers

The rewritten query will typically have more output columns than the original query. For example, a SELECT AVG will result in a SELECT SUM, COUNT. The private reader will convert the columns supplied from the rewritten query back into the shape requested in the original query. To do this, it needs to keep track of all the names used in the rewritten query, which may nest several layers deep.

Scope

The Scope object keeps track of names in each nested query, and successive layers can ask for a symbol, name pair, ensuring that names are not duplicated.

There is some arbitrary handling for anonymous expressions, which does not follow engine-specific conventions.

Pushing Aggregate SELECT Expressions

We support a limited set of differentially private expressions, such as SUM, AVG, COUNT, and VAR. The SUM and COUNT expressions are pushed through to the rewriter as SUM and COUNT expressions, while AVG and VAR are converted to formulas that use SUM and COUNT. This creates some additional work for the private reader, since the private reader needs to decide which output columns need noise, and ensure that the rewritten query is parsed to get sensitivity. This is not entirely straightforward, since numeric columns can be used in a GROUP BY statement, in which case they don't need noise, and ints may be used in SUMs or COUNTs, with differing sensitivity requirements.

It might be desirable for the rewriter to convert these output columns to ANON_SUM and ANON_COUNT, or even ANON_AVG on the outer. This would require that these grammar elements be added to the grammar, and these obviously would not work in typical engines. This is a benefit, since it would prevent the outer expressions from being executed in a SQL engine, and these expressions are intended to be handled in post-processing. All of the inner queries, which are intended to run in the engine, would remain compatible with SQL-92. This has some advantages:

- Callers could pass in their own ANON_SUM, etc. to PrivateReader to get custom differential privacy queries
- Using outer functions like ANON_AVG and ANON_VAR would allow the rewriter to more easily plug these functions to external implementations, such as the core library
- Could allow capturing of errors where outer query passes through non-noisy column. Final validation pass could ensure no non-ANON, unbounded sensitivity aggregates

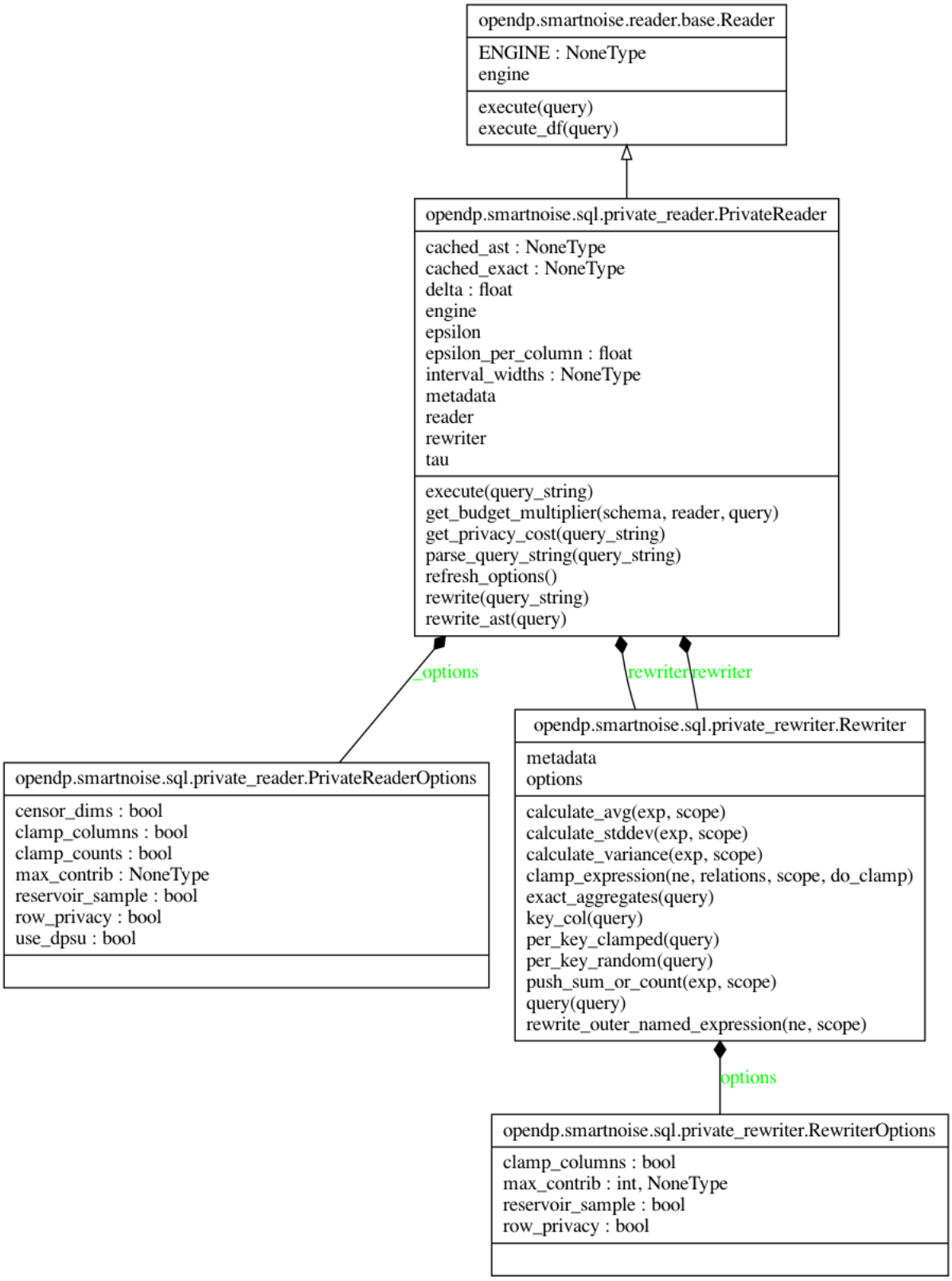
An open issue is how we handle quantiles. The current proposal is to select an underlying histogram of (e.g.) 100 items, and then use a differentially private histogram to report the requested quantile(s). This would require the rewriter to return one or more vector columns with each row.

PrivateReader

The private reader is an implementation of `SqlReader` that wraps an engine-specific reader, and performs the following steps:

1. Parse query and add symbols
2. Validate
3. Rewrite
4. Execute
5. Postprocess: add noise, evaluate formulas, filter, sort

The `from_connection` factory method is available on `PrivateReader`, and requires a `Privacy` parameter in addition to the connection and engine. Going forward, the Privacy class will be the preferred way to pass in privacy definitions.


opendp.smartnoise.reader.base.Reader
ENGINE : NoneType
engine
execute(query)
execute_df(query)
opendp.smartnoise.sql.private_reader.PrivateReader
cached_ast : NoneType
cached_exact : NoneType
delta : float
engine
epsilon
epsilon_per_column : float
interval_widths : NoneType
metadata
reader
rewriter
tau
execute(query_string)
get_budget_multiplier(schema, reader, query)
get_privacy_cost(query_string)
parse_query_string(query_string)
refresh_options()
rewrite(query_string)
rewrite_ast(query)
_options
rewriter
rewriter
opendp.smartnoise.sql.private_reader.PrivateReaderOptions
censor_dims : bool
clamp_columns : bool
clamp_counts : bool
max_contrib : NoneType
reservoir_sample : bool
row_privacy : bool
use_dpsu : bool
opendp.smartnoise.sql.private_rewriter.Rewriter
metadata
options
calculate_avg(exp, scope)
calculate_stddev(exp, scope)
calculate_variance(exp, scope)
clamp_expression(ne, relations, scope, do_clamp)
exact_aggregates(query)
key_col(query)
per_key_clamped(query)
per_key_random(query)
push_sum_or_count(exp, scope)
query(query)
rewrite_outer_named_expression(ne, scope)
options
opendp.smartnoise.sql.private_rewriter.RewriterOptions
clamp_columns : bool
max_contrib : int, NoneType
reservoir_sample : bool
row_privacy : bool

The implementation uses a functional (map/filter/sort) pattern to allow the code to run in pure Python or in PySpark. Because queries may run over very large result sets, `execute()` should return an iterator that streams records.

The rewriter returns a nested query, with the outer query not intended to execute against the database engine, but instead to be used in post-processing. This outer query returned by the rewriter should have the exact same column names and types as the original query that was passed in.

The private reader has a lot of code for handling key counts and identifiers. This is tied to the `is_key` and `is_private_key` helper methods on the AST, which relies on symbol resolution. Bugs in this code path would likely result in privacy bugs.

Privacy

The Privacy class is a property bag for setting and querying privacy information, such as epsilon, delta, alphas (for accuracy), neighboring definition, floating point protection, etc. The constructor is keyword-only, to minimize breaking changes when new capabilities are added or removed. All properties have a default value.

Accuracy

`PrivateReader` has an `execute_with_accuracy` method, which calls the normal `execute` method, and adds accuracy information for each row and column of output. Each alpha in privacy.alphas represents the probability that a noisy value falls outside of the computed accuracy range. For example, alpha of 0.05 corresponds to a 95% interval, where 95% of noisy values will fall within the range returned for accuracy.

Accuracy of SUM and COUNT is fixed for a particular epsilon, delta, and alpha. Accuracy of AVG, VAR, and STD is dependent on the noisy count, and may change for each row of output.

To support the existing pattern where rows are streamed one at a time, we need to return the accuracies with each tuple output row. Because the caller can request multiple alphas, we return the accuracies in order of alphas, wrapped in a tuple alongside the row tuple:

```
tuple(row, ((accuracies_alpha_1), (accuracies_alpha_2))
```

This supports a calling pattern like:

```
privacy = Privacy(epsilon=1.0, delta=1/1000, alphas=[0.05, 0.01])
private_reader = PrivateReader.from_connection(conn, engine="postgres",
privacy=privacy)
result = private_reader.execute_with_accuracy(query)
for row, accuracies in result:
  acc95, acc99 = accuracies
  print(row)
```

The accuracy computation needs to be done within the body of the main `execute_ast` method, when mapping output rows, so we pass an accuracy parameter with default of False through all execute methods. This flag is set to True when called from `execute_with_accuracy`. In the body of `execute_ast`, we always maintain output rows as a tuple of `(row, accuracies)`, with accuracies set to `None` when accuracy is not requested. This allows for consistent processing in the postprocessing stages of filtering, sorting, and thresholding. However, to preserve compatibility with DB-API interface when calling `execute` with no accuracy information, the execute_ast method will check to see if accuracy is False, and if so, will map all `(row, accuracies)` tuples to an iterator over rows, as the final step of processing.