

Handling of Escaped Identifiers

This design document covers handling of identifier escaping in SmartNoise SDK.

Identifiers

Identifiers in SQL-92 refer to database objects, such as databases, schemas, tables, and columns. Database names are usually case-sensitive, while schema, table, and column identifiers are case-insensitive by default. For example, a table created like

```
CREATE TABLE Foo (Col1 ... )
```

Can be queried equivalently from any SQL-92 database using syntax like:

```
SELECT * FROM FOO;  
SELECT * FROM foo;  
SELECT * FROM fOo;
```

However, if the object name uses a reserved SQL keyword, or used mixed case, it needs to be escaped. The default SQL-92 escape character is double-quotes (single-quotes are used for string literals).

```
CREATE TABLE "Select" ("Group" int, ... )
```

```
SELECT * FROM "Select";  
SELECT * FROM select; # Fails!  
SELECT "Group" FROM "Select";  
SELECT "group" FROM "select"; # OK on SQL Server, fails on PostgreSQL
```

Note that PostgreSQL treats escaped identifiers as case-sensitive, while SQL Server treats everything as case-insensitive by default.

Non-escaped identifiers must start with a character and cannot contain spaces. Escaped identifiers such as "35" and "New Column" are permitted.

Although SQL Server supports the standard SQL-92 escape sequence, identifiers can also be escaped with square brackets in SQL Server. MySQL requires backticks (`) to escape identifiers.

Note that each component of an identifier is escaped independently. So, for example, if the schema is named “My Schema” and the table is named “Group”, the query would need to specify:

```
SELECT * FROM ‘My Schema’.’Group’
```

Aliases can be escaped:

```
SELECT col_1 AS ‘My Column’ FROM FOO;
```

Search Path

SQL-92 supports fully-qualified object names like database.schema.table. Most commonly people use schema.table syntax. Most databases have a default schema, such as ‘dbo’ on SQL Server and ‘public’ on PostgreSQL. Queries that reference a naked table name are searched in a search path that typically looks in the default schema, and may be configured to search in a schema named after the logged-in user.

While it is recommended to query full schema.table names, use of naked schema is very common, and we need to be able to tell that, for example, dbo.Table1 is equivalent to Table1.

ODBC Driver Handling

In general, data access APIs pass through identifiers exactly as specified by the user. The database engine automatically resolves names and checks case-sensitivity.

The response from data access APIs typically removes escaping, because the programming language bindings just treat the column names as strings. Therefore, mapping aliases to output column names requires understanding of database identifier handling.

SmartNoise Identifier Handling

Because we do extensive identifier processing outside the database engine, there are several places where we need to ensure compatibility with various database backend behavior.

AST

The AST built in `AST.Parse` maintains full fidelity with input SQL syntax. Executing with a `DataReader` passes through with no changes, so everything works the same as when calling from the database engine's native API.

Database-specific readers know how to convert escape characters, for example to convert double-quotes to backticks, so identifiers parsed in one dialect should run without errors in another dialect.

Metadata

The `Metadata` class is used to attach type and sensitivity to nodes in the AST, via `load_symbols` on `SqlRel` relation objects. The `load_symbols` functions work by recursing relations down to terminal table references in relations. Each table name in the query is checked against the metadata file. Because the semantics of escaped identifiers vary based on database engine, the `load_symbols` function needs to adapt based on database engine, if the query uses escaped identifiers. For example:

```
SELECT * FROM 'Table1' JOIN Table1 USING(ID);
```

Will join two different tables on Postgres, but will join the same table to itself on SQL Server. Because Postgres treats these as two different tables, both tables will need metadata in the YAML file. Therefore, the YAML metadata needs to store metadata using engine-specific escaping rules. The `load_symbols` function must also handle cases where multiple expressions match the same table name. For example, the YAML might store a Postgres table name as lowercase with no escaping, such as `table1`, and both queries:

```
SELECT * FROM 'table1' JOIN Table1 USING(ID);  
SELECT * FROM Table1 JOIN Table1 USING(ID);
```

Should resolve to the same table definition in the metadata.

This means that `Metadata.Metadata`, `Metadata.ConfigFile`, and `Metadata.Database` all need to support escaped identifiers with engine-specific semantics. The calls to `SqlRel` relation objects `load_symbols` must maintain the engine-specific escaping from the

YAML, and `load_symbols` must be able to load from metadata that may not match the exact syntax the user supplied in the query definition.

Note that queries over system tables to list database objects do not escape identifiers. Therefore the database inference needs to determine if an identifier needs to be escaped before loading into metadata.

TableColumn and Column

The `TableColumn` object represents a piece of resolved metadata that has been attached as the result of a `load_symbols` call. The `TableColumn` maps to the metadata, while `Column` objects represent select expressions which may query a `TableColumn` or some transformed expression over `TableColumns`. Because the `Column` and `TableColumn` represent two different inputs (from the analyst and from the data curator, respectively), they may not always match exactly syntactically. The system must be able to perform engine-specific comparisons to determine whether a given `Column` references a given `TableColumn`.

Expressions

Expressions can be value or Boolean expressions, and represent a collection of transforms and aggregates over any number of source columns. Expressions are analyst-provided, and can appear in select statements, predicates, join criteria, and aggregates. The `symbol()` method on all `SqlExpr` object will walk down the tree recursively until resolving all `Column` references to `TableColumn` references in the associated relations.

`NamedExpressions` are a special type of expression that maps an arbitrary expression to a new name that will be available to `Column` expressions in the parent scope. The new name, called an ‘alias’, can be escaped like any other identifier. Calls to `symbol()` must be able to walk arbitrary names.

For expressions in select statements which are not provided with a name, SQL-92 defines name inference rules. For example, if a select statement selects a single column with no transformation, the new column will be named the same as the original column. Names that cannot be inferred will typically be anonymous in the parent scope, and will show some arbitrary placeholder name such as ‘???’ in ODBC result column names. Our AST allows expressions to define a `symbol_name` function which is used by the rewriter to generate friendly-looking names for complex expressions we do not wish to

be anonymous. The `symbol_name` function must be escaping-aware. For example, mapping a count expression to `'count_' + column_name` might fail, because column name might be escaped, leading to invalid identifiers with escape character in the middle of the name, like `count_[Group]`.

Expression Evaluation

All expressions can be evaluated against a dictionary of bindings, for example:

```
bindings = {
  'temperature': [50.0, 60.0, 44.0],
  'crashes': [10, 8, 3],
  'refurbished': [True, True, False]
}
```

Can be passed to an expression like “SELECT Temperature / 10, NOT Refurbished” using `expr.evaluate(bindings)`, which will result in two named expressions with anonymous names and vectors `[5.0, 6.0, 4.4]` and `[False, False, True]` respectively.

Similar to other cases with escaped identifiers, the column names used in bindings may be the result of an ODBC-style call, while the query string will be user-supplied. The bindings may include spaces or uppercase characters, and may or may not include escape characters. The system will need to use a comparer to ensure that identifiers used in the query attach to the appropriate column in the bindings.

Validator

The validator requires that symbols have been resolved, and relies on ability to determine whether or not particular columns are key columns or numeric. As long as the AST properly matches using engine-specific identifier matching, validation will work with no additional effort.

Rewriter

The rewriter works by transforming an initial query into a sequence of queries with intermediate transformations to support differential privacy.

To create each new subquery, specific transformations are created as fragmented AST expressions, and a NamingScope is established to manage name creation and avoid collisions in each child scope. The naming scope manager must be escaping-aware, both to ensure that new names use consistent escaping that looks similar to the user-supplied expressions, and to avoid name collisions where escaped and non-escaped identifiers are syntactically different but otherwise the same.

Implementation

Engine-specific escaping rules are implemented as a standard interface NameCompare object with an `identifier_match` method that can be used to compare two identifiers using engine-specific semantics. Each engine-specific implementation of DataReader provides a NameCompare object that can be attached to the query before resolving symbols.

Note that NameCompare is only necessary when resolving escaped identifiers. Because ASTs can be constructed, manipulated, and executed against database engines without ever loading symbols (and escaped identifiers will still work fine in scenarios that do not require symbols), the comparer is not always necessary.

Conversely, it may sometimes be desirable to manipulate queries with symbols and escaped identifiers without ever connecting to a database. In this case, the default comparer may be used, or the comparer from an engine-specific reader can be used without using the rest of the reader.

Supplement: Metadata Resolution Flow

This section recaps the flow of resolution for metadata.

1. `Ast.Parse` accesses a text representation of SQL. The statement is parsed with the Lexer and Parser, and then an AST is built. The system may stop at this point, if metadata processing is not desired. The AST can still be used and executed against various backend engines.
2. If metadata is supplied, the AST now walks all relations recursively to map input relations to metadata. Relations include subqueries, tables, and joins, and are represented by `SqlRel` objects. This loading process results in all relations having

a list of resolved metadata entries representing database objects accessible to that relation, including sensitivities, types, and keys.

3. If metadata was available for all relations, the parser now connects all output expressions to input metadata, by recursively walking all expressions, represented by `SqlExpr` objects. Resolution of `SqlExpr` sources must follow SQL-92 relation aliasing rules. For example, `SELECT A.Foo, B.Bar FROM X AS B JOIN Y AS A;` must ensure that the first expression searches the second relation, and the second expression searches the first relation. The output of this final pass is a list of both named and anonymous expressions at each scope containing expressions. These resolved expressions are parallel to the original AST, and reference `TableColumn` (metadata) objects rather than `Column` expressions.