

# Design Approach: SQL Subset for Differential Privacy

This document describes a subset of SQL-92 syntax and associated parser that can be used by analysts to construct differentially private queries.

We first enumerate several goals and non-goals. Our primary goal is to support a useful privacy-safe subset of SQL. Each additional language construct comes with diminishing returns and increased complexity. Therefore, we prioritize for the most useful subset that can support a broad range of backends, while aggressively limiting the completeness of language support.

## Goals

This parser must be able to run differentially private queries against most engines that support SQL-92. The parser must not rely on syntax that is peculiar to a small number of engines.

The parser must support transformation of queries into a representation that can easily target non-SQL backends using in-memory, flat row or column store representations, as described in the query semantics specification.

The parser must support a useful set of SQL-92 commonly used by analysts. For example:

- SELECT, GROUP BY, ORDER BY, WHERE, all with expressions
- Privacy-safe aggregations, like SUM, COUNT, AVG, MIN, MAX, VAR, STD
- All of the normal arithmetic and Boolean expressions, including expressions over multiple columns
- CASE statements for histograms and outputs
- Basic math functions, string and date manipulation functions

Every query which the parser accepts for differentially private processing must be valid to run with exact query semantics against the same database, independent of our parser. In other words, all valid differentially private queries are also valid non-private queries. The analyst must be able to create valid privacy-preserving queries without any syntax or semantics that are otherwise unsupported by the current database.

The parser should anticipate being ported to other programming languages and platforms, such as Scala or C++.

The parser should provide clear commitments about safety and security, including guidance about how to deploy as part of a secure system.

## Non-Goals

It is not a goal to build a tool that can generally translate queries written for one database engine to a different engine. It is not a goal to specify a subset of SQL that is guaranteed to run on all supported database engines.

The parser should not prohibit queries that happen to be idiosyncratic for a specific database engine. For example, the analyst may write a query using MySQL-specific backtick escaping, and this should not be prohibited by the parser. Conversely, it is not a goal to support all or most idiosyncrasies of each supported database backend.

Consistent with the above non-goals, the parser must not reject privacy-preserving queries which can run against SQL engines, but not easily port to non-SQL backends. In particular, many types of nested subqueries and joins in the source relation may be widely supported in SQL, and will be supported by the parser, while non-SQL backends may require flat rowset sources. It is not a goal to force queries against SQL engines to restrict sources to match semantics of all supported backend data engines.

It is not a goal to support a complete implementation of SQL-92, or even to support the maximal subset of SQL-92 that is supported by our target database engines. Many useful features such as windowing functions, union, and having clause are omitted.

## Implementation Approach

There is an important tension between our requirement to ensure all private queries are valid non-private queries, and our goal to support non-SQL backends. In particular, SQL engines have some idiosyncrasies regarding identifier escaping and subqueries and joins. Because this first requirement is non-negotiable, we need to consider this tension in our implementation approach.

The parser is built with ANTLR4, using a small subset of SQL-92. The AST parsing is initially implemented in Python, to more easily support execution engine targets of Azure ML and MLFlow, while allowing rapid iteration. As we converge on a useful and stable subset, other platforms can be supported.

We currently choose to use a single grammar for all supported SQL and non-SQL backends. An alternative approach would be to use a single grammar that supports a universal subset, and then allow augmented versions of the grammar to be used to generate engine-specific versions of the parser. We have ruled out this approach for now.

Production of differentially private queries requires a translation or “rewriting” step, where a source query is translated into something that enforces privacy rules and supports post-processing with privacy mechanism. There are two broad approaches we could use for this translation:

**Manipulate SQL Trees:** In this approach, we parse the original SQL query into an AST representation, and then modify the AST progressively to produce a new query or queries that can be used to accomplish what we want.

**Intermediate Representation:** In this approach, we parse the original SQL query into an AST representation, and then translate into an intermediate representation that loses all SQL-specific semantics, such as the intermediate representation described in the query semantics specification. This intermediate representation can then be converted to an execution plan for any given backend, with each supported backend only needing to understand the IR.

The first approach has the advantage that engine-specific syntax can be supported automatically. For example, a query using proprietary identifier escaping and complex subqueries and joins on the source relation will work automatically, because these things are unimportant to the privacy rewriter, and will simply be passed through at execution time.

The second approach has the advantage that it supports more reliable translation to different backends. The intermediate translation step forces decisions about idiosyncrasies like escaped identifiers and proprietary syntax that might otherwise be undetected and cause subtle bugs in translation.

Because SQL statements are highly structured, and because our supported query semantics are limited, much of the implementation can be agnostic to translation approach. When translating from SQL, we need to consider a handful of language components:

1. Basic language tokens, such as “select”, “where”, etc. These are limited and easy to support.
2. Value and Boolean expressions at row-level. These are simple and largely independent of any platform or language. Things like collation may impact comparison operators, and platform specific handling of numeric datatypes can lead to inconsistencies, however.
3. Aggregation and summary functions. These are limited to privacy-safe operations, which are simple and widely-supported. SQL defines semantics for handling of null values and quantifiers which may not be widely supported, however.
4. Predicates. We limit predicates to be Boolean expressions at row level, so support is the same as other expressions.
5. Relations. SQL defines complex semantics for joins, subqueries, unions, and other manipulation of relations. These cannot be easily mapped to non-SQL backends. Our query semantics require that the source rowset be presented as a single, flat rowset, so we can treat relations as a black box (apart from validation) for purposes of translation.
6. Identifiers. SQL statements essentially define a mapping between columns in source relations and output columns. Modern database engines support a variety of identifier comparison rules, including escaping of reserved words, case-sensitive and insensitive, and traversal of aliased name scopes. Support for such expressions is non-negotiable, but adds complexity that is not required in most translation scenarios.