A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Viktor Leis
Technische Universität München
leis@in.tum.de

ABSTRACT

The first contribution of the paper is a comprehensive critique of modern SQL, informed by an analysis of real-world SQL queries. This provides the motivation for our second contribution: the *Simple ANd Expressive Query Language (SaneQL)*. SaneQL features a straightforward and consistent syntax, which improves its learnability and ease of implementation. Additionally, it provides extensibility, with the added ability to define new operators that integrate seamlessly with the existing built-in ones. Unlike most data frame APIs and NoSQL query languages, SaneQL fully embraces the core principles behind SQL, especially multiset semantics. We propose that adopting SaneQL's approach can ensure the enduring success of relational database technology, offering the power of SQL's underlying concepts through a more accessible and flexible language.

1 INTRODUCTION

SQL. Despite celebrating its 50th anniversary in 2024 [4], SQL is still *the* predominant query language. Its success is inextricably linked with the success of the relational model. In comparison with other languages, it stands out for its declarative nature, multiset semantics, and three-valued logic. These concepts have stood the test of time and enable data independence, effective query optimization, and automatic parallelization.

Problem 1: Irregular Pseudo-English Syntax. Unusually among widely-used programming languages today, SQL has an English-inspired syntax. The motivation behind the surface syntax of SQL stems from the desire to make queries easy to read. SQL's inventor Don Chamberlin calls this the "walk up and read" property [2]. It is true that the meaning of a query like `SELECT name FROM customer WHERE id = 42` can indeed be guessed without any formal training. However, this is only true for simple queries, and optimizing for this kind of readability comes at great cost. The irregularity of modern SQL, which has grown tremendously over the past five decades, makes it hard to learn, cumbersome to write, difficult to debug, hard to implement, and leads to impenetrable error messages. The dominance of SQL may be at risk, as evidenced by the rising popularity of data frame APIs, such as Python's Pandas. Such APIs have, to a significant extent, already replaced SQL in data science.

Problem 2: Lack of Extensibility and Abstraction. The cornerstone of any powerful programming language is a mechanism for abstraction – and SQL is lacking severely on this front. SQL offers views (and their transient variant Common Table Expressions) but these merely allow naming and re-using parts of a query. For example, it is not possible to pass a relation into a view definition as argument. This effectively makes SQL a functional programming language that has functions *without* parameters. More advanced features such as passing expressions are obviously not supported either. For example, imagine implementing a semi join or pivot operator that works for arbitrary input relations and join predicates. This is simply not possible in SQL. SQL fundamentally violates the "don't repeat yourself" principle in software engineering, and instead relies on sheer repetition.

Contributions. This paper makes two contributions. First, Section 2 provides a detailed critique of modern SQL. We do this through example queries that highlight SQL's weaknesses, and through the analysis of real-world SQL queries that show the difficulties SQL learners face. Our critique provides the rationale for the second contribution, the *Simple ANd Expressive Query Language (SaneQL)*, which we introduce in Section 3. SaneQL has a simple and regular syntax, which not only makes it easier to learn and implement, but also enables extensibility. In SaneQL, it is possible to define new operators that are indistinguishable from built-in ones. In contrast to most data frame APIs and NoSQL query languages, SaneQL fully embraces the core ideas behind SQL, in particular multiset semantics. We believe that SQL has become successful due to its powerful underlying concepts rather than its unusual syntax – and that adopting a new modular surface syntax is the best way to ensure the enduring success of relational database technology.

2 A CRITIQUE OF MODERN SQL

Query Dataset. In this section, we critique modern SQL through query examples that illustrate its irregular nature. Additionally, we provide empirical evidence for the claim that SQL is hard to learn through a real-world query dataset. We collected 130,998 queries from the https://hyper-db.com website in January 2019. The website provides a web-based SQL interface and is primarily used by students learning SQL at the Technical University of Munich (TUM) and other universities. The final exam for the TUM *Introduction to Databases* undergraduate course, which has over 1,000 students, takes place in February. Thus, the dataset reflects the difficulties that SQL learners preparing for an exam have, rather than the difficulties experienced users face. The first striking result is that out of all queries, 50,683 (38%) result in an error when executed. The vast majority of these errors are compile time (not runtime) errors. While some of these are clearly unavoidable (e.g., incomplete queries), we believe that many cases would be unnecessary with a simpler, more regular query language.

Table 1: Error messages of 50,683 invalid SQL queries from https://hyper-db.com website.

%	error message (in PostgreSQL)
39.8	syntax error at ...
16.6	column ... does not exist
9.0	column ... must appear in the GROUP BY clause
7.2	relation ... does not exist
5.7	missing FROM-clause entry for table
4.0	subquery in FROM must have an alias
2.2	division by zero (runtime error)
2.2	column reference ... is ambiguous
1.9	aggregate functions are not allowed in ...
1.5	operator does not exist
1.4	each UNION query must have same number of columns
1.2	function does not exist
0.9	invalid reference to FROM-clause entry for table
0.6	SELECT * with no tables specified is not valid
0.5	aggregate function calls cannot be nested

2.1 Syntax

Unhelpful Error Messages. Table 1 shows the most common error messages as reported by PostgreSQL. Many of the most common error messages are not very helpful, and usually do not explicitly say what the conceptual problem is or how to fix it. This is particularly true for *syntax errors*, which, as Table 1 shows, are by far the most common error type. As we discuss below, the underlying root cause is often irregular grammar leading to spurious commas and misplaced parentheses.

WITH. A construct that is surprisingly difficult to get right is Common Table Expressions (CTEs). Semantically, Common Table Expressions without recursion are straightforward: they define one or more transient views that can then be referenced like any base table. When teaching SQL, we recommend using the `WITH` construct to decompose larger queries into intermediate steps. The correct syntax is as follows:

```
WITH r AS (SELECT 1), s AS (SELECT 2) ...
```

The following variants are illegal (and it's easy to construct more):

```
WITH r AS (SELECT 1) s AS (SELECT 2) ...
WITH r AS SELECT 1, s AS SELECT 2 ...
WITH (r AS SELECT 1), (s AS SELECT 2) ...
WITH r AS (SELECT 1), s AS (SELECT 2), ...
WITH r AS (SELECT 1) WITH s AS (SELECT 2) ...
```

We have encountered many similar errors in our query dataset. Overall, only 45% of all queries containing `WITH` are correct. Finally, as an example of SQL's limited syntactical orthogonality, observe that in the context of WITH one has to write `WITH name AS value`, while in the `SELECT` clause one writes `SELECT value AS name`.

WITH RECURSIVE. SQL also offers recursive common table expressions, which appear as a minor syntactic variation of non-recursive CTEs, but, by making the language Turing complete, are much more powerful. The following example computes the Fibonacci sequence:

```
WITH RECURSIVE fib(a,b) AS
 (SELECT 0, 1 -- base case
  UNION ALL
  SELECT b, a+b -- recursive case
  FROM fib
  WHERE a<100)
SELECT a FROM fib
```

In most programming languages, recursion is a powerful and general construct. SQL's recursive CTEs, on the other hand, are highly restricted: the SQL standard only permits referencing the recursive relation ("fib" in the example) once, which prevents many interesting computational use cases [1]. This limitation implies that the construct is semantically equivalent to iteration rather than recursion. Indeed, the best way to understand the semantics of a WITH RECURSIVE query is through the actual iterative execute logic of the construct: it first executes the base case, then iteratively executes the "recursive" case using the prior result as input until it returns an empty result. We argue that specifying this process as a recursion often makes formulating complex computational logic unnecessarily abstruse and hard to understand.

VALUES. Another syntactically difficult SQL construct is the construction of constant inline tables, which uses the following syntax:

```
SELECT * FROM (VALUES ('x',5), ('y',2)) AS r(a,b)
```

Note the specific locations of the four pairs of parentheses. The following three variants result in a syntax error:

```
SELECT * FROM VALUES ('x',5), ('y',2) AS r(a,b)
SELECT * FROM (VALUES ('x',5), ('y',2) AS r(a,b))
SELECT * FROM (VALUES (('x',5), ('y',2))) AS r(a,b)
```

Given this unforgiving syntax, we are not surprised that students find it difficult to write correct VALUES expressions. In our dataset only 40% of all queries that contain the `VALUES` construct are correct.

Verbose Base Table Queries. The simplest (and maybe most common) query is probably `SELECT * FROM table`. This raises the question why it is not sufficient to simply write `table`. A related question is why one has to write `SELECT * FROM r UNION SELECT * FROM s` rather than `r UNION s`.

Too Many Keywords. The SQL parser relies heavily on reserved keywords. SQL-92 defined 227 keywords, and over time this number increased, arriving at 409 with SQL:2023[1]. These 401 keywords include many common English terms, and represent 18% of all English word usage[2]. It is therefore not surprising that, in our query dataset, a significant fraction of invalid table and attribute accesses stem from reserved keywords being used as identifiers. The SQLite documentation comments on this situation as follows: "The list of keywords is so long that few people can remember them all. For most SQL code, your safest bet is to never use any English language word as the name of a user-defined object."[3]

[1]https://www.postgresql.org/docs/current/sql-keywords-appendix.html
[2]This was computed using English word frequency data from https://wortschatz.uni-leipzig.de/en/download/English.
[3]https://www.sqlite.org/lang_keywords.html

2.2 Joins

Implicit Cross Products. Some may argue that SQL's syntactical issues are merely annoyances, so let us next turn our attention to more fundamental issues. The relational model favors the use of normalized tables. As a consequence, many SQL queries contain joins, which makes a good syntax for joins important for any relational language. The oldest [4] and most common construct for inner joins is relying on implicit cross products:

```
SELECT r_regionkey, r_name
FROM region, nation
WHERE r_regionkey = n_regionkey
```

This syntax is concise and mathematically elegant, but it has a major downside: in complex queries it is easy to forget a join predicate. This will introduce an unintentional cross product, which can have disastrous consequences including system overload. Such a bug is particularly easy to miss when the query contains an aggregate. For example, the following query will take a long time and then produce a subtly wrong result:

```
SELECT o_orderpriority, avg(l_quantity)
FROM nation, customer, orders, lineitem
WHERE n_nationkey = c_nationkey
AND o_orderkey = l_orderkey
AND n_name = 'GERMANY'
GROUP BY o_orderpriority
```

Given that in practice intentional cross products are rare, syntactically representing such a dangerous operator through a comma may not be the most ergonomic choice. This choice is even more striking given the fact that SQL is generally very verbose – with cross products being a dangerous exception.

Explicit Join Trees. SQL also offers an explicit syntax for joins in the form of `FROM r JOIN s on r.a=s.b`. While for inner joins, this is merely syntactic sugar, the explicit syntax is the only way to express outer joins. The conceptual reason for this is that the semantics of outer joins depends on the syntactically specified join order. The following two expressions, for example, may produce different results:

```
FROM r LEFT JOIN s ON r.a=s.b LEFT JOIN t ON s.c=t.d
FROM r LEFT JOIN s LEFT JOIN t ON s.c=t.d ON r.a=s.b
```

The reason for potentially different results is that the location of the `ON` clause determines the ordering of the two joins: the former query represents the order $(r⟕s)⟕t$, while the latter means $r⟕(s⟕t)$. SQL's syntax makes this semantic distinction hard to see.

2.3 Aggregation

Redundant Aggregation Attributes. Another common operation is grouping and aggregation:

```
SELECT r_regionkey, r_name, count(*)
FROM region, nation
WHERE r_regionkey = n_regionkey
GROUP BY r_regionkey, r_name
```

Note that the grouped attributes appear twice: once in the `SELECT` and once in the `GROUP BY` clause. This behavior is also surprising to SQL learners, with 9% of all queries failing with a "column must appear in the GROUP BY clause" error.

HAVING. Another fairly common error, which occurs in 1.9% of all erroneous queries, is to filter an aggregate function in the `WHERE` clause. Although this would be possible without a special syntax using a subquery and `WHERE`, SQL offers the `HAVING` clause:

```
SELECT r_regionkey, r_name, count(*) c
FROM region, nation
WHERE r_regionkey = n_regionkey
GROUP BY r_regionkey, r_name
HAVING count(*) > 4
```

Thus, even though SQL promises to be declarative, there are implicit ordering dependencies between the different syntactic clauses. More on this observation later.

WITHIN GROUP. Ordered-set aggregate functions, which require specifying a sort order, rely on the following baroque syntax:

```
SELECT o_custkey, percentile_cont(0.5)
          WITHIN GROUP (ORDER BY o_totalprice)
FROM orders
GROUP BY o_custkey
```

We do not see any good reason why the `WITHIN GROUP` qualifier is required for ordered-set aggregates. Since all aggregate functions are computed "within their group", why is it necessary to announce this so prominently for some of them? The mere fact that they require sorting does not change its semantics, and sorting is a common implementation technique for normal aggregates as well. The only explanation is historical contingency and the need to avoid parsing issues caused by an English-like syntax.

2.4 Window Functions

OVER. A window function computes a new attribute based on other tuples of the input relation, which is useful for time series analysis, ranking, top-k, percentiles, moving averages, and cumulative sums [11]. SQL relies on the following syntax:

```
SELECT o_custkey, rank() OVER (ORDER BY o_totalprice)
FROM orders
```

Again we are confronted with a new keyword (`OVER`). Even taking "walk up and read" at face value, one may wonder what over indicates in the context of computing a rank.

Filtering. In SQL, window functions are computed quite late (after `HAVING` but before `ORDER BY`). A consequence of this is that it is not possible to access window functions in the `WHERE` clause. Thus, a common operation such as filtering based on a rank requires a subquery and becomes quite cumbersome:

```
SELECT o_custkey, rk
FROM (SELECT o_custkey,
             rank() OVER (ORDER BY o_totalprice) rk
      FROM orders)
WHERE rk < 4
```

Notably, SQL provides the `GROUP BY` clause with its own filtering mechanism via the `HAVING` clause, yet window functions lack a

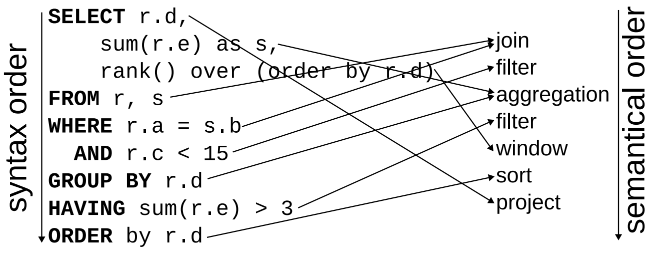

Figure 1: Illustration of SQL syntax clauses and their semantical evaluation order.

similar feature. This asymmetry seems arbitrary as there appears to be no conceptual justification for this inconsistency.

Operator Ordering. Another consequence of the fact that window functions are executed after aggregation is that while it is possible to access aggregates in window functions (e.g., `rank() OVER(ORDER BY count(*))`), applying an aggregate on a window function again requires a subquery:

```
SELECT o_custkey, max(rk)
FROM (SELECT o_custkey,
            rank() OVER (ORDER BY o_totalprice) rk
      FROM orders)
GROUP BY o_custkey
```

The examples illustrate that the execution order of window functions relative to the other operators is quite arbitrary and not visible in the query itself. One simply has to memorize the fact that they come after aggregation, and then introduce subqueries as needed. Thus, SQL is not as declarative as it appears. In fact, it relies on implicit ordering rules that have to be memorized to understand the semantics of non-trivial queries.

2.5 Syntactic Order Is Not Semantic Order

As we already saw in several examples, the order of SQL's syntactic clauses does not correspond to their semantical evaluation order. Consider the query shown on the left-hand side of Figure 1. To write this query correctly, one has to remember the syntax order of each clause (1. `SELECT`, 2. `FROM`, ...). However, this is not enough: to obtain the desired query semantics, one also has to know the semantical execution order of each clause. As the arrows in the figure illustrate, the syntax order prescribed by SQL has little relationship to the semantical execution order. The `SELECT` clause may be the most egregious example: its components can be executed at very different points in time. A further complication is that, as we discussed earlier, sometimes this order does not coincide with one's intention and one is forced to introduce subqueries to explicitly impose a semantical order. Thus, although SQL is often called a declarative language, it relies on subtle semantical ordering constraints that are not visible in its syntactic structure.

2.6 Lack of Portability

SQL has been called an "intergalactic standard" and its weaknesses might be forgivable if it would at least ensure portability across different systems – alas, this is not the reality. Although SQL is an official ISO standard, in practice, it is very hard to write portable SQL queries. There are several reasons for this: Even though the SQL standard is voluminous, it leaves the semantics of many basic constructs undefined. One example is static type inference: Assuming attribute `a` is an integer, what is the result type of `a/2`? Furthermore, many systems choose to deviate from the standard. For example, 5 out of 14 tested database systems do not support the standard infix string concatenation operator ||[4]. Finally, all widely-used systems implement non-portable expression libraries and language extensions, which further fragments the language. Thus, SQL is not a single, well-defined language, but a set of mutually-intelligible, but incompatible dialects.

2.7 Lessons

Irregular Syntax Causes Big Problems. We have demonstrated that using English-inspired syntax leads to a complex and arbitrary grammar, making the language difficult to learn and often resulting in impenetrable error messages. Such a syntax also makes the language hard to extend, as every new feature requires new syntax. As the SQL standard has grown through the decades it accumulated more and more complexity. The fundamental lesson from this is that languages have to be designed with extensibility and abstraction mechanisms in mind.

Semantic Operator Ordering Should Be Explicit. We also demonstrated that in order to compose accurate SQL queries, it is necessary to comprehend the implicit ordering semantics of each construct. We therefore argue that it would be better for query languages to make the semantical and syntactical order identical. Therefore, we believe that (extended) relational algebra with its explicit operator ordering is a good foundation for a modern query language. Note that to teach SQL one generally first teaches relational algebra anyway, as this allows explaining each operator individually. Also note that this does not mean that the explicitly specified order also corresponds to the execution order (as is done by many data frame APIs). After all, query optimizers work at the relational algebra level anyway.

3 SANEQL: TOWARDS SIMPLE AND EXPRESSIVE QUERIES

Motivation. To address the shortcomings of SQL we propose a new query language, called the *Simple ANd Expressive Query Language (SaneQL)*. Its benefits are two-fold: First, it provides a nicer and more systematic way to expressive queries, which resembles the well known data frame APIs. But while this is clearly an improvement over the syntax oddities of SQL, this alone may not be sufficient to justify a new query language. The second, and perhaps even more important aspect of SaneQL is modularity. SaneQL allows reusing logic across queries, which is not traditionally done in SQL besides the very limited capability offered by views. In the following, we first look at the basic functionality, which can be seen as a nicer SQL, and then explain how meta-programming works in SaneQL, which goes beyond what can be expressed in SQL alone.

[4]https://github.com/sqlstandardsproject/sqlacidtest

```
filter(r table, condition expression)
join(left table, right table, on expression,
     type symbol := inner)
as(r table, name symbol)
map(r table, expressions list expression)
project(r table, expressions list expression)
projectout(r table, remove list symbol)
group(r table, by list expression, aggs list expression,
     type symbol := group, sets list list symbol := {})
order(r table, by list expression)
union(left table, right table)
unionall(left table, right table)
intersect(left table, right table)
except(left table, right table)
distinct(r table)
window(r table, expressions list expression,
       partitionby list expression := {},
       orderby list expression := {},
       framebegin expression := unbounded(),
       frameend expression := currentrow(),
       frametype symbol := values)
iterate(base table, recursion expression,
        increment symbol := iterate)
```

Figure 2: Relational operators

3.1 Basics

Foundation: Relational Algebra. Conceptually, SaneQL is based on relational algebra. Similar to the explicit join syntax in SQL, the user can describe queries by combining relational operators, operating on tables and scalar values. And just like with regular SQL, the initial algebra expression is then transformed by the query optimizer into a better execution plan. For expressiveness, SaneQL introduces three new categories of values that do not exist in SQL namely *expressions*, *symbols*, and *lists* of expressions and symbols. We will come back to the exact definition of these later when talking about modularity, but as first approximation an *expression* is an unevaluated term such as a join condition, and a *symbol* is name of a column, relation, or operator.

Pipelining Through UFCS. Syntactically, SaneQL expresses everything as nested function calls with (optionally) named parameters. Similar to the programming languages D and Nim, SaneQL supports Uniform Function Call Syntax (UFCS), which means that we can use a dot to pass a value as first argument of the next call. Thus, a.join(b,...).groupby(...) is a different, more convenient way to write groupby(join(a,b,...),...). The dot notation is usually preferable as it preserves locality in the query text, which is very convenient when formulating queries. Similar to chaining of Unix commands though pipes, in SaneQL queries are usually formulated by appending operators step-by-step at the end of the query.

Syntax: Calls, Keywords, and Lists. Consider the following example computing the revenue for every customer from Germany:

```
nation.filter(n_name='GERMANY')
.join(customer, c_nation=n_nationkey)
.join(orders, o_custkey=c_custokey, type := leftouter)
.group({o_custkey}, {revenue := sum(o_totalprice)})
```

The example demonstrates several features. First, relational operations are performed by invoking functions on tables. A list of functions defined on tables is shown in Figure 2. Note that these built-in functions are in no way special, and we can define new functions that work just the same in SaneQL, as we will demonstrate below. Second, named parameters help to keep complex invocations readable and the function signature self-documenting, for example the join type is specified as left outer join here. And third, curly braces are used to denote lists: in the example both for the grouping expressions and for the aggregate functions. Inside the curly braces we can use the named-value-syntax to assign names to newly computed columns, here to `revenue`. The result of that query is a table with two columns `o_custkey` and `revenue`.

Scoping. When accessing a table, the system implicitly puts a tuple variable with the same name into the scope, which we can use to disambiguate column names. If a tuple is referenced multiple times `as(...)` can be used to rename an intermediate result. We demonstrate both concepts in the following query:

```
y1.join(y2, y1.x=y2.x).join(y2.as(t3), y2.y=t3.y)
```

Inline Tables. Constant tables are constructed with a `table` call:

```
r.join(table({{y:=1, color:='red'}, {2, 'blue'}}), x=y)
```

Let Construct. To simplify query formulation one can name intermediate results with the `let` construct. This corresponds to a `WITH` statement is SQL, but supports both scalars and tables. Just like `WITH`, a `let` is query-local. We can make it persistent by using `define` instead, which corresponds to `CREATE VIEW`. `let` is very convenient for structuring a query, as shown below:

```
let year := 2023,
let rev_in_year := orders.filter(o_year=year)
    .group({o_custkey}, {total:=sum(o_totalprice)}),
customer.join(rev_in_year, type:=leftouter)
```

3.2 Modularity and Extensibility

Scalar Arguments. The basic examples we have seen so far could all be expressed in SQL. What makes SaneQL much more expressive is the ability to parameterize queries, which is the basis for modular query formulation. For example, we could have formulated the last example with a scalar parameter:

```
let rev_in_year(year) := orders.filter(o_year=year)
    .group({o_custkey}, {total:=sum(o_totalprice)}),
customer.join(rev_in_year(2023), type:=leftouter)
```

Expression Arguments and Correlated Subqueries. Even more useful than scalar parameters are table parameters, as they allow us to formulate queries that work in very different scenarios. For example, we could implement a generic `avg_revenue` function and use it to compare the revenues across very different kinds of customers:

```
let avg_revenue(p expression) := customer.filter(p)
    .join(orders, o_custkey=c_custkey)
    .group({o_custkey},{total:=sum(o_totalprice)})
    .aggregate(avg(total)),
let avg_building := avg_revenue(c_mktsegment='BUILDING'),
let avg_regular := avg_revenue(c_comment.like('%reg%'))
```

Expression arguments can also be used to express correlated subqueries, as the following translation of TPC-H query 2 illustrates:

```
let min_cost_for_part(p_partkey) :=
    partsupp.filter(ps_partkey=p_partkey)
    .join(supplier, s_suppkey=ps_suppkey)
    .join(nation, s_nationkey=n_nationkey)
    .join(region.filter(r_name='EUROPE'),
          n_regionkey=r_regionkey)
    .aggregate(min(ps_supplycost)),
part
.filter(p_size = 15 && p_type.like('%BRASS'))
.join(partsupp, p_partkey = ps_partkey)
.join(supplier, s_suppkey = ps_suppkey)
.join(nation, s_nationkey = n_nationkey)
.join(region.filter(r_name='EUROPE'),
      n_regionkey=r_regionkey)
.filter(ps_supplycost=min_cost_for_part(p_partkey)) ...
```

Evaluation Rules. SaneQL distinguishes between a scalar parameter (the default), a table parameter (indicated by `table`) and an expression parameter (indicated by `expression`): Scalar and table parameters are evaluated in the scope of the caller, and the result is then passed to the called function, while expression parameters are passed unevaluated and are then evaluated in the scope of the called function when the expression is evaluated. Expression arguments are similar to lambda functions from other programming languages, except that they can intentionally access the full scope when they are invoked.

Semi Join Example. This mechanism allows for expressive queries. Let's pretend that SaneQL did not offer semi-joins and we want to implement them ourselves. In reality it does offer semi-joins, by setting `type:=leftsemi` when calling join, but for illustrative reasons we want to do that manually and we want to do this by using a regular join and then removing duplicates. Note that the preserved table contains duplicates, which we have to preserve, and we do not know anything about the join condition. Nevertheless, we can formulate a generic semi-join query as follows:

```
define leftsemi(preserve table, probe table, p expression)
  let x := gensym(),
      preserve
      .window({x := row_number()})
      .join(probe, p)
      .project({probe, x})
      .distinct()
      .projectout({x})
```

`gensym` creates a unique symbol `x`, which can be used to safely add a column to an intermediate result. This mechanism is similar to macros in Common Lisp and avoids potential name conflicts that could arise when simply adding a column with a fixed name. Then, it stores a unique integer in the column `x`, which makes the input duplicate free, performs the join with the probed table, removes duplicate join results, and finally removes the column `x` from the result. While a native semi-join is clearly more efficient, this demonstrates the expressiveness of the language, as after defining this function once, it works for arbitrary tables and arbitrary join conditions. Due to the UCS, this function can be used like a built-in table operation. Thus, when our `leftsemi` function is in scope, the following will just work: `rel1.leftsemi(rel2, x=y)`.

Summary. These mechanisms allow us to formulate queries using reusable components, where frequently occurring constructs are implemented once and then used with different parameters instead of formulating everything over and over again as typically done in SQL. This is a huge improvement in usability and finally brings query languages closer to what we are used to in regular programming languages.

3.3 Implementation

We designed SaneQL to be a native query language. Due to its simple syntactic structure and straightforward mapping to relational algebra, it is significantly easier to implement than a SQL frontend. Our prototype implementation, which is meant for illustrative purposes, translates SaneQL to SQL. The source code of the translator and 22 TPC-H queries is available at https://github.com/neumannt/saneql.

4 RELATED WORK

How We Got Here. Codd proposed not just the relational model itself [5], but also a declarative query language based on first order predicate calculus called ALPHA [6]. In ALPHA, queries are specified through explicit tuple variables that are connected through predicates and universal or existential quantifiers. The query language of Ingres, QUEL [14], is based on ALPHA, but avoids the use of quantifiers while providing more general aggregation capabilities. The roots of SQL are the Structured English QUEry Language (SEQUEL), proposed by Chamberlin and Boyce in 1974 [4]. SEQUEL can be thought of as providing a more friendly, non-mathematical syntax for predicate calculus through the SELECT-FROM-WHERE-GROUP BY clauses. It also includes support for subqueries for expressing universal or existential quantifiers. The original SEQUEL proposal is conceptually elegant and none of the issues identified in Section 2 apply – but the functionality of the language is quite limited. SEQUEL 2 [3] therefore added support for HAVING, UNIQUE (now called DISTINCT), ORDER BY, and NULL (but not yet outer joins). The SQL-86 and SQL-89 standards are quite close to SEQUEL 2, and the next major change was the influential SQL-92 standard which became the least common denominator supported by virtually all relational database systems. It adds support for outer joins and allows subqueries in almost every part of the query. The SQL:1999 [9] and SQL:2003 [10] standards then incorporated common table expressions, window functions, and ordered-set aggregate functions – resulting in what we call *modern SQL*. Tracing this language evolution over five decades, the addition of each individual feature seems both rational and beneficial. Yet, since the initial language design was not designed with extensibility in mind, each new feature demanded the creation of new syntax. The result is a hard-to-learn language of unnecessary complexity.

SQL Critiques. SQL has been criticized for almost as long as it exists. The criticism can be categorized into two classes: lack of orthogonality and deviations from the relational model. Date's 1984 critique [8] focuses on the former. Some of the major points he raised, such supporting subqueries in the `FROM` clause, have been addressed in later SQL versions. Codd [7], on the other hand, criticizes SQL for its use of multiset (rather than set) semantics, its reliance on subqueries, and the lack of support for four-valued logic. We believe that SQL's semantics has stood the test of time, and advocate for a more orthogonal and modular surface syntax.

New Query Languages. There have been many attempts at a better query language. Popular ones include Microsoft's Language Integrated Query (LINQ)[5] and Kusto Query Language (KQL)[6] and data frame APIs such as Python's Pandas[7], R's dplyr[8], and Rust's/Python's Polars[9]. These approaches have in common that they are tightly integrated into the host programming language rather than the DBMS. Data frame APIs appear similar to relational algebra, but their semantics (e.g., ordered and deterministic results, named rows, eager operator evaluation) make automatic query optimization and parallelization challenging, leading to suboptimal query performance [12]. This is a major challenge for a project like Modin [13], which strives to achieve high performance in a backwards-compatible way. We argue that database systems should implement an algebra-based language that adopts the underlying concepts of SQL but provide an orthogonal and simple language interface. SaneQL is our attempt at such a language, bringing innovations of data frame APIs and in particular dplyr's pipelining construct to database query languages. Through its `define` construct, SaneQL also offers the capability to implement higher-order abstractions within the language itself. This distinguishes SaneQL from languages like Pipelined Relational Query Language (PRQL)[10], which otherwise share many of the same ideas.

5 SUMMARY AND FUTURE WORK

The Problems of SQL. Despite being the dominant query language for half a century, SQL has major weaknesses. For beginners, it is not only hard to learn and difficult to debug due to impenetrable error messages, but its irregular syntax also makes common convenience features such as auto-completion and built-in interactive documentation very challenging. For expert users, SQL's verbosity and lack of abstraction capabilities slow down ad hoc interactive data analytics. Additionally, the fact that the standard does not specify important semantical issues such as implicit type casting rules makes porting queries across database systems a gamble. Finally, for developers of database systems, the complexity of the language makes it a major undertaking to implement a SQL frontend. This indirectly makes developing new systems much more difficult than necessary and therefore slows down progress in the data management field.

SaneQL. The root cause of most of the problems enumerated above is SQL's irregular and unusual pseudo-English syntax. We argue that a simple regular syntax together with an explicit ordering of relational operators is a better foundation for a modern query language. We therefore propose the SaneQL language, which is easy to learn and implement. It relies on the same underlying semantics as SQL, which makes SaneQL easy to offer it as alternative query language to existing systems. The language is also extendible, i.e., one can define new operators that are indistinguishable from built-in ones.

Future Work. While we designed SaneQL as a user-facing language, it can also be useful in other cases, which we plan to investigate in the future. SaneQL's syntactical elements closely resemble those of many modern scripting languages, allowing for its direct embedding into a host language A version of the language with additional operator hints could also be used to represent physical query plans. Making the query language and the optimized plans similar would make it easier for users to debug performance problems and to inject explicit query plans. We also believe that the step-by-step semantics of SaneQL makes it possible to implement powerful and intuitive interactive query construction interfaces, a topic we plan to investigate further. Finally, although SaneQL already offers advanced abstraction capabilities, there are abstractions that would require a general macro system.

ACKNOWLEDGMENTS

We would like to thank Maximilian Kuschewski for insightful discussions on the new language and contributions to the implementation. Funded/Co-funded by the European Union (ERC, CODAC, 101041375). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. 2022. Machine Learning, Linear Algebra, and More: Is SQL All You Need?. In *CIDR.*
[2] Donald D. Chamberlin. 2023. 49 Years of Queries. In *SIGMOD Keynote.*
[3] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. 1976. SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. *IBM J. Res. Dev.* 20, 6 (1976), 560–575.
[4] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *SIGMOD Workshop, Vol. 1.* ACM, 249–264.
[5] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.
[6] E. F. Codd. 1971. Data Base Sublanguage Founded on the Relational Calculus. *Research Report / RJ / IBM / San Jose, California* RJ893 (1971).
[7] E. F. Codd. 1990. *The Relational Model for Database Management, Version 2.* Addison-Wesley.
[8] C. J. Date. 1984. A Critique of the SQL Database Language. *SIGMOD Record* 14, 3 (1984), 8–54.
[9] Andrew Eisenberg and Jim Melton. 1999. SQL: 1999, formerly known as SQL 3. *SIGMOD Rec.* 28, 1 (1999), 131–138.
[10] Andrew Eisenberg, Jim Melton, Krishna G. Kulkarni, Jan-Eike Michels, and Fred Zemke. 2004. SQL: 2003 has been published. *SIGMOD Rec.* 33, 1 (2004), 119–126.
[11] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *PVLDB* 8, 10 (2015), 1058–1069.
[12] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *PVLDB* 13, 11 (2020), 2033–2046.
[13] Devin Petersohn, Dixin Tang, Rehan Sohail Durrani, Areg Melik-Adamyan, Joseph Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. *PVLDB* 15, 3 (2021), 739–751.
[14] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. 1976. The Design and Implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (1976), 189–222.

[5] https://learn.microsoft.com/en-us/dotnet/csharp/linq/
[6] https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query/
[7] https://pandas.pydata.org/docs/user_guide/index.html
[8] https://dplyr.tidyverse.org/
[9] https://www.pola.rs/
[10] https://github.com/PRQL/prql