SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL

Jeff Shute
Google, Inc.

Shannon Bales
Google, Inc.

Matthew Brown
Google, Inc.

Jean-Daniel Browne
Google, Inc.

Brandon Dolphin
Google, Inc.

Romit Kudtarkar
Google, Inc.

Andrey Litvinov
Google, Inc.

Jingchi Ma
Google, Inc.

John Morcos
Google, Inc.

Michael Shen
Google, Inc.

David Wilhite
Google, Inc.

Xi Wu
Google, Inc.

Lulan Yu
Google, Inc.

sql-pipes-paper@google.com

ABSTRACT

SQL has been extremely successful as the de facto standard language for working with data. Virtually all mainstream database-like systems use SQL as their primary query language. But SQL is an old language with significant design problems, making it difficult to learn, difficult to use, and difficult to extend. Many have observed these challenges with SQL, and proposed solutions involving new languages. New language adoption is a significant obstacle for users, and none of the potential replacements have been successful enough to displace SQL.

In GoogleSQL, we've taken a different approach - solving SQL's problems by extending SQL. Inspired by a pattern that works well in other modern data languages, we added piped data flow syntax to SQL. The results are transformative - SQL becomes a flexible language that's easier to learn, use and extend, while still leveraging the existing SQL ecosystem and existing userbase. Improving SQL from within allows incrementally adopting new features, without migrations and without learning a new language, making this a more productive approach to improve on standard SQL.

PVLDB Reference Format:
Jeff Shute, Shannon Bales, Matthew Brown, Jean-Daniel Browne, Brandon Dolphin, Romit Kudtarkar, Andrey Litvinov, Jingchi Ma, John Morcos, Michael Shen, David Wilhite, Xi Wu, and Lulan Yu. SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL. PVLDB, 17(12): 4051 - 4063, 2024.
doi:10.14778/3685800.3685826

1 INTRODUCTION

SQL has been tremendously successful, standing the test of time for 50 years[17] as the primary language for structured data processing, supported near-universally across databases and query tools.

SQL is not an easy language to learn or use. Even for expert users, SQL is challenging to read, write and work with, which hurts user productivity. Several alternative languages have been proposed, but none have gained widespread adoption or displaced SQL. Migrating away from existing SQL ecosystems is expensive and generally unappealing for users.

This paper presents a different approach. After describing the most critical problems with the SQL language, we present a solution – adding pipe-structured data flow syntax to SQL. This makes SQL more flexible, extensible and easy to use. This paradigm works well in other languages like Kusto's KQL[5] and in APIs like Apache Beam[1]. We show pipe syntax can be added to SQL too, without removing anything, and while maintaining full backwards compatibility and interoperability.

In SQL, the standard clauses occur in one rigidly defined order. Expressing anything else requires subqueries or other workarounds. With pipe syntax, operations can be composed arbitrarily, in any order. This increases flexibility, radically simplifies the user experience, and enables clean language extension.

For example, standard SQL cannot express multi-level aggregations without subqueries, resulting in queries with complex "inside-out" data flow. This is query 13 from the TPC-H benchmark:

```
SELECT c_count, COUNT(*) AS custdist
FROM
  ( SELECT c_custkey, COUNT(o_orderkey) c_count
    FROM customer
    LEFT OUTER JOIN orders ON c_custkey = o_custkey
         AND o_comment NOT LIKE '%unusual%packages%'
    GROUP BY c_custkey
  ) AS c_orders
GROUP BY c_count
ORDER BY custdist DESC, c_count DESC;
```

With pipe syntax, equivalent logic can be expressed sequentially, applying operators top-to-bottom in arbitrary orders.

```
FROM customer
|> LEFT OUTER JOIN orders ON c_custkey = o_custkey
      AND o_comment NOT LIKE '%unusual%packages%'
|> AGGREGATE COUNT(o_orderkey) c_count
   GROUP BY c_custkey
|> AGGREGATE COUNT(*) AS custdist
   GROUP BY c_count
|> ORDER BY custdist DESC, c_count DESC;
```

Stonebraker writes in [13] (via [15]):

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685826

> My biggest complaint about System R is that the team never stopped to clean up SQL… All the annoying features of the language have endured to this day. SQL will be the COBOL of 2020, a language we are stuck with that everybody will complain about.

It's been 50 years. **It's time to clean up SQL.** This paper shows one promising approach.

1.1 GoogleSQL

We've implemented pipe syntax in GoogleSQL[19], the SQL dialect and implementation shared across all[1] SQL systems at Google including F1[28], BigQuery[2], Spanner[19] and Procella[18], and the open-source release, ZetaSQL[12]. GoogleSQL is a shared, reusable component, enabling many systems to share the same SQL dialect. This shared component allowed implementing pipe syntax in one place and then enabling it across many products.

2 EVALUATING STANDARD SQL

2.1 Syntax problems

Many critiques of SQL have been written, from 1984[21] through 2024[25]. [15] describes many challenges in detail.

Fundamentally, the SQL language is difficult for users. For beginners, SQL is hard to learn. Even for expert users, SQL is awkward and annoying – it's hard to write, hard to read and hard to edit.

Here, we describe several syntax issues. (Pipe syntax addresses all of these!)

2.1.1 Clause order. SQL's challenges start from its basic query syntax: `SELECT ... FROM ... WHERE ... GROUP BY`, etc. This operation[2] order is rigid and arbitrary and doesn't reflect the actual data flow, which starts with table scans in the `FROM`. Figure 1 illustrates this disconnect.

While common, this operation order is far from universal, and expressing any other ordering requires using subqueries. Reordering specific clauses could improve readability somewhat, but wouldn't address the limited expressivity.

2.1.2 Redundant clauses. SQL works around rigid clause order in particular cases by adding duplicate clauses, which increases language complexity. To filter results before and after aggregation, and after window functions, SQL uses `WHERE`, `HAVING` and `QUALIFY`. These all express the same filtering operation, in different places, with different syntaxes, and with subtly different rules and behaviors.

2.1.3 Need for subqueries. Many simple operations can only be expressed using subqueries:

- Filtering anywhere other than the three supported locations.
- Aggregating two or more times.
- Projecting computed expressions before the final `SELECT` so they can be referenced multiple times by name, in later `SELECT` items, `WHERE` clauses, `JOIN`s, etc.
- Using queries as table-valued function (TVF) inputs.

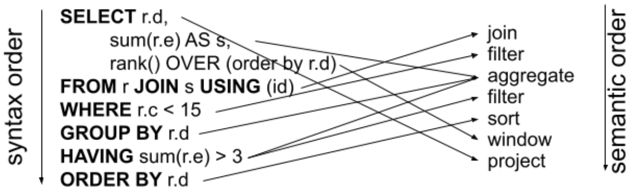

Figure 1: SQL syntactic clause order doesn't match semantic evaluation order. (From [25].)

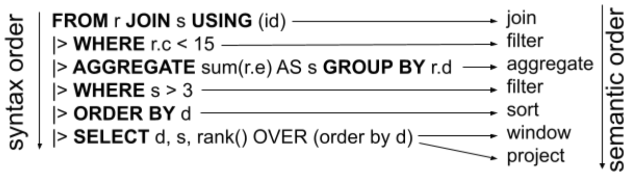

Figure 2: In pipe syntax, operator order matches semantic evaluation order. (Execution order will still be optimized.)

Subqueries require indentation to reflect nesting clearly, which hurts readability. Editing SQL typically involves frequent refactoring, wrapping query fragments into subqueries, indenting, and adding repetitive `SELECT` and `GROUP BY` clauses, before the desired operation can finally be added.

2.1.4 "Inside-out" data flow. With the inline `FROM` clause, data flow in standard SQL starts in the middle with the most deeply nested table reference. Then logic builds outwards, applying operations both above and below the starting point, while traversing outwards through layers of nested subqueries.

This "inside-out" structure makes tracing through SQL logic difficult. In large queries, it can be hard to even find the starting point.

WITH clauses help, but don't fully solve this. `WITH` clauses (defining *Common Table Expressions* or *CTEs*) can help split up queries, reducing nesting and moving initial logical blocks earlier in the query text, but CTE solutions have several caveats:

- There is significant boilerplate per CTE subquery – at least "`WITH ...` CTE_k `AS (SELECT * FROM` CTE_{k-1}`)`".
- The CTEs and the final query each still have inside-out data flow starting from their local `FROM` clause, referencing an earlier CTE by name. It's still not possible to read the whole query top-to-bottom.
- Referencing CTEs by name is more verbose and complex than sequential pipe operators. It's necessary to read carefully to understand CTEs are actually sequential, with each CTE_k referencing just CTE_{k-1}.
- Adding, removing or reordering operators in a CTE sequence requires updating name references in other CTEs or the main query.
- CTEs are too verbose to fully expand with one CTE per operation, so CTEs themselves are often non-trivial subqueries. (Pipe operators are naturally 1:1 with relational operations and can directly express any operation sequence.)

[1] Except CloudSQL hosted third-party databases and Postgres-compatible AlloyDB.

[2] Terminology: We use *operation* as in relational algebra operation, generalized to include the query engine's supported logical operations. We use *operator* to describe particular syntactic clauses in the query, including pipe operators.

2.1.5 Side-effects at a distance. Even locally within single queries, behavior is complex and non-local, and syntax is repetitive. SELECT and GROUP BY are thoroughly intertwined, and aggregation modifies the behavior of other clauses. The same columns are cross-referenced in SELECT, GROUP BY and ORDER BY, and corresponding edits are often required in three places.

2.1.6 Poor extensibility. Extending SQL with new query operations is difficult. TVFs are a powerful extension mechanism but the call syntax makes TVFs awkward to actually use since they compose poorly with built-in operations and other TVFs, requiring more nested subqueries as a workaround. (More in 4.3.)

Summary. These syntax issues combine to make SQL difficult to learn, and excessively difficult to work with, even for SQL experts.

2.2 Besides the syntax, SQL is great!

SQL's foundational semantics work very well. The relational data model, where everything is a table and relational operations consume and produce tables, provides excellent composability, allowing users to factor logic arbitrarily with views, TVFs, temp tables, and subqueries. Subqueries also support natural transitions between expressions and tables. While compound data types weren't considered originally, SQL extends cleanly to support structured data types like arrays, structs, protocol buffers[28] and JSON[26].

Declarative semantics differentiate SQL from many other languages and APIs, and make SQL particularly well-suited for expressing query workloads. SQL, and relational algebra, clearly separate the syntactic and semantic description of **what** to compute from the implementation details of **how** to compute it. Query optimization is fundamental in SQL and a key enabler for performance, efficiency and scalability, decoupling logical requests from physical storage, query execution strategies, and optimizations like indexing.

Finally, SQL is pervasive. SQL works in almost all database and query systems, making basically all data available through SQL somewhere, and joinable across systems using federated engines like F1 or BigQuery. Many front-ends, business intelligence tools, data modeling layers, object-relational mappings and other tools are built assuming a SQL backend.

This extensive ecosystem and large userbase make SQL sticky. Adopting non-SQL systems is more difficult and leads to fragmentation across the user's data ecosystem, where other SQL systems are likely still used.

Most alternative-to-SQL languages are lacking in some of these areas. None have the ecosystem or existing userbase, and many are lacking the foundational semantics or composability to express relational algebra fully. Declarative semantics are often lacking, sometimes because alternative platforms don't have the same optimization capabilities, forcing users to entangle business logic and execution strategy.

2.3 Why not create a better language?

It seems possible to create a new language that keeps the good properties of SQL, while offering better syntax and usability. Why hasn't this happened yet?

Actually, maybe it has. Many alternative "better than SQL" languages have been proposed (e.g. [11],[25]). None have become mainstream or widespread solutions that seem like SQL replacements, for several reasons:

- SQL is a huge language with many features, accumulated over decades. New languages are unlikely to offer feature parity, and will often be missing some needed features.
- A new language requires an excellent implementation before it can be practically used. Existing SQL products benefit from decades of engineering.
- A new language isn't useful until users learn to use it well.
- New languages usually lack interoperability and composability with existing SQL, making incremental adoption difficult. Code gets siloed between old and new languages, until there's some hard cut-over after some large migration.[30]

The challenges that come with learning and adopting a new language, and migrating to it, make it a difficult choice for an organization to adopt an alternative language. Without provable large improvements, it's hard to commit to migration. Large migrations are notoriously expensive, slow and failure-prone.[23]

Creating a new language with SQL parity often means building on the same relational algebra fundamentals, supporting at least the same features. The resulting language may effectively be SQL but with different syntax, making it insufficiently different to justify learning a new language and moving to new tools.

Some challenges are mitigated if the new language is implemented as a proxy that rewrites into SQL. That allows a new language to reuse an excellent existing SQL implementation. Challenges remain getting users productive in a new language, migrating existing code to a new tool, and dealing with limited interoperability between the new language and legacy SQL. Proxied rewriters also suffer some practical issues – they introduce additional production dependencies which may compromise latency or reliability, they don't feel like first-class APIs in the underlying systems, and they make debugging correctness and performance difficult when query that actually runs is decoupled from what the user wrote.

Outside the SQL context, conventional wisdom is that it takes a decade for a new language to succeed and become mainstream – consider C -> C++, C++ -> Rust, Java -> Kotlin, JavaScript -> Dart, etc.[30]

2.4 Our alternative: Fixing SQL syntax

Given SQL's problems, and the challenges involved with replacing SQL, we propose an alternative approach: Fixing SQL.

We assess SQL at various levels:

(1) SQL's foundational semantics, from relational algebra, are excellent.
(2) SQL's conceptual data model and top-level syntax, with statements representing queries, DDL, DML, etc, and composability via subqueries, works well.
(3) SQL's basic operations within a query (clauses like JOIN, ORDER BY, etc), all work reasonably well.
(4) SQL's syntactic structure for composing queries using those operations is **terrible**.
(5) SQL's localized syntax with English-like keyword phrases is an anachronism, but we can live with it.
(6) SQL's expression language is fine, and implementations typically include a good library of existing functions.

So we will just fix item #4, leaving the rest as is. Fortunately, this is easy to do. As we've seen in other languages, pipe-structured data flow works well. We simply reuse the existing query clauses and operations from SQL, preserving their existing syntax where practical, and recast them as independent operators that can be composed arbitrarily into queries using pipe-structured data flow.

This gives us an easy-to-use variant of SQL, which requires minimal new learning since it uses all the same concepts and mostly the same syntax. The data model and top-level statements (`CREATE VIEW`, etc) are identical, the concepts and operations are the same, and the expression language is identical. The query operation syntaxes are modified but familiar, and they now compose in a simple and intuitive way.

Section 4 describes our language design in more detail.

3 RELATED WORK

Chamberlain describes the origin and evolution of SQL in [16], including the original design goal to have a "walk up and read" property, motivating SQL's pseudo-English syntax. This was a significant tradeoff since it meant SQL was not a "functional" or composable language, as would be expected and preferred by more technical users. This design choice has aged poorly as SQL's complexity has increased over time, beyond the core language from SQL-92.

Many critiques of SQL have been written (see 2.1). [25] is a good example, describing several challenges similar to those we describe. Then it proposes a new alternative language, SaneQL, which like our proposal, cleanly expresses data flow through relational operations. Like other SQL alternatives, this proposal would require users to learn a new language and migrate away from their SQL ecosystem, which are significant obstacles.

PRQL[11] is another recent alternative language, implemented as a front-end that translates to SQL. It also provides composable relational operations, but the unfamiliar syntax and detachment from SQL make adoption challenging.

Our work contrasts with SaneSQL and PRQL by solving the same problems inside SQL, rather than by replacing SQL and hoping users will adopt new tools. The concepts we're using are not particularly novel, but applying them inside SQL is new.

SQL++[26] also extends and modernizes SQL, but is primarily focused on improving support for structured data types like JSON, without addressing SQL's core syntax problems.

Python DataFrame APIs like Pandas[9] have become popular. For many users, Python and data frame APIs are more usable than SQL, especially for ad hoc data exploration and wrangling. These APIs also offer operator chaining with pipe-like data flow. Data frame implementations typically execute imperatively within a Python process, so they don't get the declarative semantics, scalability and performance of SQL systems. While data frame APIs are popular, they fill slightly different niches, and are unlikely to replace SQL.

Others have argued that SQL is too low-level and should be replaced by higher-level languages or APIs (e.g. [6],[7]). Higher-level languages work well in many contexts, but don't replace SQL. Typically, they are implemented as front-ends that generate SQL. Pipe SQL syntax is also useful here, to simplify generating SQL, and to make generated SQL easier to understand and debug.

User-friendliness is less important in the context of generated queries, but humans still read and write SQL frequently. Many across the industry see value in improving SQL usability[24][27] and like Google, have been adding incremental usability features like `GROUP BY ALL`[4]. Pipe syntax offers a more transformative usability improvement.

[29] chronicles repeated attempts over many decades to replace SQL and the relational model, and how the industry keeps returning to SQL when alternatives fail. It argues this will continue for decades more, and new ideas won't become mainstream until they're available in SQL. Pipe syntax makes SQL and the relational model extensible (see 4.3), facilitating experimentation and quick adoption of new features and paradigms. It's not necessary to invent new languages and systems to prove out new concepts first if extending SQL directly is easy.

4 PIPE SYNTAX IN SQL

4.1 Syntax

Any query can have zero or more pipe operators as a suffix, delineated with the pipe character "`|>`". Each pipe operator starts with an operator name (one or more keywords) followed by its own argument grammar. Many of these operators reuse the existing grammar for standard SQL clauses. Operators can be applied in any order, any number of times.

Additionally, we make standalone `FROM` clauses valid queries, which can be followed by pipe operators to build arbitrary queries.

The basic grammar is:

```
<query> :=
   {all existing query syntaxes}
   | "FROM" <from_body>          -- New: FROM as a query
   | <query> "|>" <pipe_operator>  -- New: pipe suffixes

<pipe_operator> :=
   "WHERE" <expression>
   | "ORDER" "BY" <order_by_body>
   | "JOIN" <table_expression> [["AS"] alias]
   ...
```

Figure 2 shows how this aligns syntax with semantics.

Figure 3 shows the pipe operators added in GoogleSQL to achieve parity with standard SQL queries.

4.1.1 Projection operators. We define multiple projection operators for convenience:[3]

```
FROM orders
|> SET o_orderstatus = LOWER(o_orderstatus)
|> EXTEND ROUND(o_totalprice) AS dollars
|> DROP o_comment
|> AS result
|> SELECT o_custkey, result.dollars
```

- `SELECT` produces a new table with exactly the listed columns, like the outermost `SELECT` in a table subquery. Standard features like `SELECT DISTINCT` and `SELECT *` are also supported.

[3] Except where noted, example queries use TPC-H tables. They are runnable in ZetaSQL[12] using the `execute_query` tool.

Operator	Output (modified rows and columns)
SELECT <expr> [[AS] alias], ...	Same rows, with exactly these selected columns. DISTINCT, * and other common SELECT modifiers are also allowed here.
EXTEND <expr> [[AS] alias], ...	Same rows, with added columns.
SET <column> = <expression>, ...	Same rows, with updated values for modified columns.
DROP <column>, ...	Same rows, minus dropped columns.
AS <alias>	Same rows and columns, but with a new table alias available. Old table aliases are removed.
WHERE <condition>	Subset of input rows passing condition.
LIMIT <n> [OFFSET <m>]	Subset of input rows subject to counts, respecting order if input table was ordered.
AGGREGATE <agg_expr> [[AS] alias], ...	Full-table aggregation, with one row, with a column for each aggregate expression.
AGGREGATE [<agg_expr> [[AS] alias], ...] GROUP BY <grouping_expr> [AS alias], ...	Aggregation with grouping, with one row per group. The column list has the grouping columns and then the aggregate columns. Aliases can be assigned directly on grouping expressions.
[LEFT|...] JOIN {table/subquery/...} [ON <condition> | USING(col, ...)]	Join results, with a filtered cross-product of the pipe input table and the table expression following JOIN.
ORDER BY <expr> [ASC|DESC], ...	Same rows but sorted.
{UNION|INTERSECT|EXCEPT} {ALL|DISTINCT} (<query>), (<query>), ...	Set operation output, including combined rows from the input table plus *one or more* tables passed as arguments.
CALL tvf(args, ...)	Table-valued function output rows and columns. The pipe input table is passed to the TVF as its first table-typed argument.
TABLESAMPLE <method> (<size> {ROWS|PERCENT})	Subset of rows produced by the chosen sampling algorithm.
PIVOT (agg_expr FOR col IN (value1, ...))	New table with rows pivoted to become columns.
UNPIVOT (value_col FOR key_col IN (col1, ...))	New table with columns pivoted to become rows.

Figure 3: Pipe operator syntax and behavior

- EXTEND propagates the existing table, adding additional computed columns, similar to SELECT *, new_column. EXTEND also preserves table aliases, which can be necessary for disambiguation (e.g. WHERE t1.x = t2.x).
- DROP removes columns, similar to SELECT * EXCEPT (column) in GoogleSQL.
- SET replaces column values, similar to SELECT * REPLACE (expression AS column) in GoogleSQL.
- AS introduces a table alias for the current row, which is occasionally useful.

DROP and SET aren't strictly necessary but are convenient. EXTEND is useful to project values without losing existing columns or table aliases. For example, EXTEND can compute new columns between JOINs, while preserving existing table aliases.

Unlike standard SQL, projection operators can be applied repeatedly to incrementally compute expressions without (non-standard) lateral column references. Those computed columns are then visible to use in operators like WHERE. The SQL optimizer should still combine these computations as appropriate.

```
FROM lineitem
|> EXTEND l_quantity * l_extendedprice AS cost
|> EXTEND cost * l_discount AS discount
|> WHERE discount > 1000
|> AGGREGATE SUM(cost), SUM(discount)
```

These projection operators are allowed[4] to compute window functions (with OVER), but cannot do aggregation. Aggregation is a standalone operator, which simplifies syntax and semantics and preserves operator independence.

4.1.2 Aggregation.

Full-table aggregation is expressed with a list of aggregate columns to compute.

```
FROM orders
|> AGGREGATE SUM(o_totalprice) AS price, COUNT(*) AS cnt
```

Aggregation with grouping is expressed by adding a GROUP BY as part of the AGGREGATE operator[5].

```
FROM orders
|> AGGREGATE SUM(o_totalprice) AS price, COUNT(*) AS cnt
   GROUP BY EXTRACT(year FROM o_orderdate) AS year
```

The output table contains the grouping columns (which can be computed expressions, with assigned aliases) if any, and then the aggregate columns. GROUPING SETS, etc also work here.

[4] Rationale: We originally had a separate WINDOW operator but found using it inconvenient. Projecting scalar and window functions at the same time is common. Another awkward pattern occurred when users compute a window function just to use it in WHERE or ORDER BY and then need to drop the computed column.

[5] Rationale: We used the same operator name for full-table and grouped aggregation to minimize edit distance between these operations. Unfortunately, this puts the grouping and aggregate columns in different orders in the syntax and output. Putting GROUP BY first would require adding a required keyword before the AGGREGATE list.

Unlike standard SQL where columns must be repeated between the `SELECT` and `GROUP BY`, pipe `AGGREGATE` is a single standalone operator that achieves both. Expression matching (between `GROUP BY` and `SELECT`, including grouping by ordinals or aliases), isn't needed since the grouping expressions occur just once.

Since `AGGREGATE` already produces exactly the grouping and aggregate columns, it's often unnecessary to write a `SELECT` in queries that includes pipe `AGGREGATE`.

4.1.3 Filtering. The pipe `WHERE` operator can be added anywhere, including after `AGGREGATE`. Separate operators aren't needed for `HAVING` and `QUALIFY`.

4.1.4 Why have a pipe character? People ask why not just allow writing operator clauses in arbitrary orders without using a pipe character. Some other languages have this structure without using pipe characters – notably GQL[3], the SQL-adjacent Graph Query Language.

The pipe character is useful for several reasons.

Familiarity and precedents:

(1) Unix pipes are a widely understood paradigm that helps make this syntax and behavior familiar and easy to understand.
(2) Other data processing languages (e.g. KQL) use pipe syntax, and it works well for their users.
(3) Pipe-like dataflow is common in more imperative data processing APIs like Pandas DataFrames and Apache Beam.

Technical reasons:

(1) SQL parsing depends heavily on reserved keywords (which can't be used as unquoted identifiers). Many clauses can end with an optional identifier or alias, which cause parsing ambiguities with unreserved keywords. Using a pipe character clearly delineates pipe operators without requiring every operator to begin with a reserved keyword.
(2) Simple parsing facilitates language extensibility. See 4.3.
(3) Adding pipe operators on the end of queries in standard syntax is clearer when using a pipe character, making it obvious where pipe operators begin. Many pipe operators reuse the names of corresponding standard clauses, so it could be ambiguous whether they are intended as pipe operators or standard SQL clauses.
(4) Pipe operators have slightly different behaviors than the corresponding standard SQL clauses. In particular, pipe operators can only see columns from the immediate input table.
(5) Standard SQL clauses don't work as standalone operators or in arbitrary order. It's misleading to use syntax that appears to allow that. For example, standard `SELECT` and `GROUP BY` are co-dependent and have side-effects on other clauses.

Aesthetic and readability reasons:

(1) It can be useful to identify at a glance whether a query is using pipe operators, and where it transitions from standard syntax.
(2) Some pipe operators have multiple clauses (like `AGGREGATE ... GROUP BY`) which are part of the same operator, but are naturally formatted on multiple lines. Without pipe characters, they can be misread as independent operators.
(3) Pipe characters make the syntax more obviously structured, and visually splittable as a sequential list of N operations. It seems useful for the syntax to reflect this logical query structure.

As a thought exercise, if it was possible to parse statements without semicolon separators, or `SELECT` lists or functions calls without commas between arguments, would those syntaxes be preferable?

4.1.5 Why use ‘|>’ for the pipe character? The most natural and obvious choice would be to use ‘|’. Unfortunately, in GoogleSQL (and many other SQL dialects), ‘|’ is already used for bitwise OR, and reusing it for pipe operators causes parsing ambiguities. e.g.,

```
FROM Part
| SELECT *, p_size+1
| EXTEND p_type
| SELECT p_name
| AGGREGATE -COUNT(*)
```

could be parsed as

```
FROM Part
| SELECT *, (p_size+1 | extend) AS p_type
| SELECT (p_name | aggregate) - COUNT(*)
```

This claim on ‘|’ is unfortunate since bitwise math is used rarely in SQL queries, while pipe operators will be used frequently. It would be preferable to use the better syntax for pipe operators, relegating bitwise OR to an alternate syntax.

To avoid this conflict, we used ‘|>’, which is also used for similar purposes in other languages (including JavaScript[10], F#, R, and OCaml).

We explored deprecating existing usage and reclaiming ‘|’ for pipe syntax. In existing usage, we saw ‘|’ used mostly within parenthesized expressions (where it would be unambiguous), or where simple lookahead heuristics (for numeric literals or parentheses) could resolve most cases. Such heuristic solutions could allow overloading ‘|’ for both purposes, avoiding most breaking changes, but these approaches do not seem robust. More drastic solutions requiring users to migrate queries did not seem desirable.

4.2 Data model and semantics

Intermediate (and final) results in SQL are tables. A table has one or more columns, optionally with names, and zero or more rows. Relational operations produce a table as output, and SQL queries resolve as a tree of relational operations.

With pipe syntax, a query starts with a nullary relational operation, which produces an initial table. These syntaxes can be used to start a pipe query:

- Any standard-syntax SQL query (`SELECT ... FROM ... WHERE ... GROUP BY`, etc).
- A standalone `SELECT` clause without `FROM`, which produces a single row.
- `TABLE t`, which produces a full table scan of one table.
- Any standard-syntax `FROM` clause, written alone.

The first three items are supported as queries already. The last item is new, and allows queries to start with a `FROM` clause. The `FROM` clause can contain table scans, joins, TVF calls, array UNNEST, VALUES, and other common syntaxes to produce an initial result table.

Zero or more pipe operators can follow that initial query fragment. Each pipe operator is a unary relational operation that takes one table as input and produces one table as output. Most operators take additional arguments to specify exactly what to compute. Some operators (like JOIN) accept table-typed arguments (possibly subqueries), effectively making them binary (or N-ary) relational operations. All pipe operators treat the pipe input table as the primary table input, and produce one pipe output table.

4.2.1 Name scoping. Intermediate tables have a visible schema describing an ordered list of columns, optionally with names, reflecting what SELECT * would produce for that table. Additionally, there can be hidden columns (called pseudo-columns in GoogleSQL, which are selectable by name but are not included in SELECT *) and table aliases (which can be dereferenced using alias.column_name). The name scope for an intermediate table describes the visible columns, plus these hidden columns and table aliases.

Each pipe operator is a self-contained operation that can see only its input table and its arguments, and can only resolve names from the scope associated with the pipe input table. (Inside subqueries, correlated references to names from outer queries are also allowed.) Pipe operators have no visibility to earlier or later operations in the same query, so complex scoping or data flow is impossible by construction. This makes pipe operators naturally composable, so they can be applied in any order, any number of times.

4.2.2 Defining pipe operators. Each pipe operator can be defined by specifying:

- What arguments does it take (with what syntax)?
- How does it affect columns? (More precisely, what's in its output name scope?)
- How does it affect rows?
- Does it preserve order?

Most operators fit in one of these categories, depending how they affect input columns:

(1) Operators that pass through input columns unchanged:
    e.g. WHERE, ORDER BY, LIMIT
(2) Operators that produce an entirely new (operator-defined) column list:
    SELECT – produces a new table with the specified columns
    AGGREGATE – produces a new table with the grouping and aggregate columns
    CALL – produces a new table with a TVF's output schema
(3) Operators that augment or modify the input column list:
    EXTEND, DROP, SET – projection operators that modify the column list
    JOIN – extends the input table with columns and a table alias for the joined table

Order preservation property is used with ORDER BY and LIMIT. As in standard SQL, table scans and most operations produce unordered tables. ORDER BY makes its result ordered. Specific operators (primarily projection operators like SELECT) preserve order. If LIMIT is applied on an input table with order, it selects the top-N rows. Otherwise, LIMIT selects N rows arbitrarily.

4.2.3 Declarative semantics. Queries with pipe syntax still have declarative semantics. Pipe syntax appears to imperatively describe the computation as sequential steps, but the syntax specifies declarative semantics only, not an execution strategy. As in standard SQL, queries are typically converted to an algebraic form and then optimized, such that they run faster but produce equivalent results, "as if" the query was run as written. ("Equivalence" allows for ordering differences and other artifacts that can result from reordering joins, using indexes, etc.)

4.2.4 Pipe operators vs relational operations. Unlike standard SQL, where resolving a query to relational operations is complex, pipe operators correspond basically 1:1 with relational operations. Simple relational operations can always be expressed as one pipe operator.

In a few cases, for user convenience, pipe operators can expand to multiple sequential relational operations. For example, SELECT DISTINCT produces a projection and then an aggregation. Projection operators like SELECT can include window functions, and thus expand to a projection followed by windowed aggregation. Other operators that accept expressions may introduce a projection before the main operation.

This duality between relational algebra and SQL syntax is useful and does not exist with standard SQL. Bi-directional translation is possible – SQL generators no longer need to introduce complex transformations that make generated SQL difficult to read. Optimized query plans could be converted back into equivalent pipe SQL explaining clearly how a query will actually run. Query engines might support a feature to execute a pipe query as written, without optimization.

4.3 Extensibility

4.3.1 Extensibility with table-valued functions. Table-valued functions (TVFs) are a powerful extension mechanism in SQL, effectively allowing users to add arbitrary relational operations.

TVFs with only scalar arguments are nullary relational operations, which can be called in FROM clauses to produce tables dynamically. With SQL TVFs, these are effectively parameterized views. Non-SQL TVFs can produce arbitrary input tables.

TVFs that take at least one table argument are unary or N-ary relational operations. Non-SQL TVFs with table arguments can extend SQL with arbitrary new relational operations.

The major caveat when extending SQL using TVFs is that the syntax is too awkward. For example, BigQuery uses TVFs to express ML model lookups[31], which can look like:

```
SELECT *
FROM ML.PREDICT(
    MODEL `my_project.imdb_classifier`,
    (
      SELECT *
      FROM ML.PREDICT(
        MODEL `my_project.nnlm_embedding_model,`,
        (SELECT '<text>' AS input, 7 AS rating))
    )
  )
```

With pipe syntax, these TVFs can be invoked directly using the CALL operator without using nested subqueries, like this:

```
SELECT '<text>' AS input, 7 AS rating
|> CALL ML.PREDICT(MODEL `my_project.nnlm_embedding_model`)
|> CALL ML.PREDICT(MODEL `my_project.imdb_classifier`)
```

Any TVF can still be called with the usual syntax in FROM or JOIN, but TVFs that take table arguments can also be called with the pipe CALL operator. The provides near-first-class syntax for operators implemented as TVFs.

A standard TVF that can be invoked like

```
FROM some_tvf((SELECT ...), arg2, arg3, ...)
```

can be invoked with pipe CALL like this:

```
SELECT ...
|> CALL some_tvf(arg2, arg3, ...)
```

The TVF's first table-typed argument is treated like a this argument (in object-oriented call style), and implicitly receives the pipe input table. Other arguments are written with the usual argument syntax.

This allows invoking TVFs in a natural way, expressing the logical order of computation without excessive nesting and subqueries. The difference is particularly stark when considering nested or chained TVF calls, which require deeply nested subqueries in standard syntax (as in the example above), but look like flat sequential calls in pipe syntax.

TVF CALLs with pipe syntax are as composable as any pipe operator, so they can be used like language extensions. CALLs are limited to using function call syntax, but optional named arguments and structured types allow significant flexibility. (A potential extension could allow TVFs to provide a plug-in grammar snippet, allowing TVFs to have first-class syntax too. Grammar extensions are impractical in standard SQL, but one pipe operator can be parsed independently between successive pipe characters.)

4.3.2 Extensibility for built-in operators. Standard SQL is difficult to extend and evolve, even for built-in operations. Many operations don't fit naturally anywhere in standard SQL grammar. New operations must be forced into the SELECT ... FROM ...GROUP BY flow somewhere, which usually involves complex syntactic and semantic interactions.

Standard SQL's pseudo-English syntax depends heavily on reserved keywords to parse unambiguously. Adding new syntax without new reserved keywords is often difficult, or requires opportunistically reusing existing reserved keywords in novel ways. New reserved keywords are always breaking changes for some potential existing queries, making such additions difficult.

This may partly explain why SQL has evolved slowly, and why new functionality sometimes has awkward syntax.

For example, consider PIVOT[20] (which is non-standard but widely supported), which produces multiple columns with slices of aggregate values taken from multiple rows, "pivoting" rows to become columns. For example, this query:

```
SELECT *
FROM (
    SELECT n_name, c_acctbal AS bal, c_mktsegment
    FROM customer JOIN nation ON c_nationkey = n_nationkey
  ) PIVOT(SUM(bal) AS bal
         FOR n_name IN ('PERU', 'KENYA', 'JAPAN'))
```

behaves like a shorthand for:

```
SELECT
  c_mktsegment,
  SUM(IF(n_name = 'PERU', bal, NULL)) AS bal_PERU,
  SUM(IF(n_name = 'KENYA', bal, NULL)) AS bal_KENYA,
  SUM(IF(n_name = 'JAPAN', bal, NULL)) AS bal_JAPAN
FROM
  (SELECT n_name, c_acctbal AS bal, c_mktsegment
   FROM customer JOIN nation ON c_nationkey = n_nationkey)
GROUP BY c_mktsegment
```

In standard SQL, there's no natural place to put the PIVOT operator, so it's kludged in as a suffix on subqueries in the FROM clause. PIVOT acts on all columns of the input table, so it's almost always necessary to use a subquery to prepare the input table, and it's always required to have more query syntax around PIVOT, at least doing SELECT *. So PIVOT queries are always complicated.

For language implementers, without a uniform framework for adding operators, PIVOT is a complex special case in the grammar, that can attach after tables, subqueries, or TVFs, with complex interactions with other suffix clauses like TABLESAMPLE. These interactions can trigger subtle bugs so tests must cover all combinations.

Now contrast PIVOT as a pipe operator. While the operation is exactly the same, using exactly the same syntax inside the PIVOT clause, it fits naturally as a pipe operator, making it easy to call as a fully composable operator that can be parsed, implemented and tested in isolation.

And it's easy for users to call, without subqueries and with minimal wrapping. Syntax before and after PIVOT is only needed if there's useful logic to express.

```
FROM customer JOIN nation ON c_nationkey = n_nationkey
|> SELECT n_name, c_acctbal AS bal, c_mktsegment
|> PIVOT(SUM(bal) AS bal
        FOR n_name IN ('PERU', 'KENYA', 'JAPAN'))
```

Similar challenges exist adding other operations added to SQL, both in the standard and as product-specific extensions. Some examples:

- MATCH_RECOGNIZE[8] (from SQL:2016) adds a mini-language for analyzing sequential patterns. Like with PIVOT, there's no natural place to add the syntax, so it gets added as another table suffix inside FROM, requiring subqueries before and after to prepare input and consume output.
- [14] proposes adding stream processing in SQL. It proposes using TVFs since that makes the operations composable with clean semantics, but then suffers from the syntactic complexities of using TVFs.

New operations like these make more sense as pipe operators, since pipe syntax makes them both easier to use and easier to specify and implement. Pipe operators are naturally composable and have locally defined syntax (no need for new reserved keywords!) and semantics. This potentially unlocks significant innovation in SQL functionality and usability that is too difficult in standard SQL.

Pipe operators proposed so far aim for parity with existing SQL syntaxes, but novel new operators might be supported with pipe syntax only, maybe with TVF syntax as a fallback.

4.3.3 Some experimental extensions: Debugging operators. Figure 5 shows some additional pipe operators we have prototyped as language extensions, going beyond what can be expressed in standard SQL.

Debugging SQL can be challenging (see also 5.3) compared to imperative languages where control flow is obvious in the code. These operators facilitate many common debugging patterns by making it easier to extract or validate intermediate state in a SQL query with simple local syntax additions.

These examples show how pipe syntax enables novel language extensions without any complex interactions with other SQL syntaxes. New pipe operators can always be used anywhere in a query, while they often wouldn't fit well anywhere inside standard SQL.

4.4 Interoperability and composability

Since pipe syntax is implemented as a feature within the existing GoogleSQL dialect, the new syntax is fully interoperable and composable with standard syntax. Any query can be written in purely standard syntax, standard syntax plus pipes, or fully with pipes starting from `FROM`. Subqueries can be written in either syntax, inside queries written in either syntax. Views and TVFs can be declared using either syntax. DDL and DML statements containing queries can use either syntax.[6]

This allows users to adopt pipe syntax incrementally, where they find it useful, while continuing to use any existing SQL code they have. Users might write new queries in pipe form, or convert to pipe form when updating or debugging existing queries. Where refactoring (adding table subqueries, etc) would be needed anyway, users might instead refactor into pipe form, making further logic changes easier.

Supporting incremental adoption, without requiring migration or bifurcating SQL codebases, is a major advantage over alternate-language proposals. Supporting the new syntax inside query engines, without a separate translation proxy, provides full interoperability automatically, including for example, calling views bidirectionally.

5 EVALUATION

This section covers our experience using GoogleSQL with pipe syntax, including several use cases that benefit.

5.1 Usage at Google

After an initial implementation phase, iterating on the language design with a small group of early users, we stabilized the language and made pipe syntax available to users broadly inside Google.

Over the following six months, we've seen adoption and usage increase steadily – see figure 4. The two initial spikes follow announcements sharing pipe syntax on a SQL users mailing list. The second followed removing opt-in settings and making pipe syntax available by default. After initial usage spikes from users trying our demo queries, we see many users continue to use pipe syntax as part of their daily work, with further usage growth as the feature spreads virally to more users.

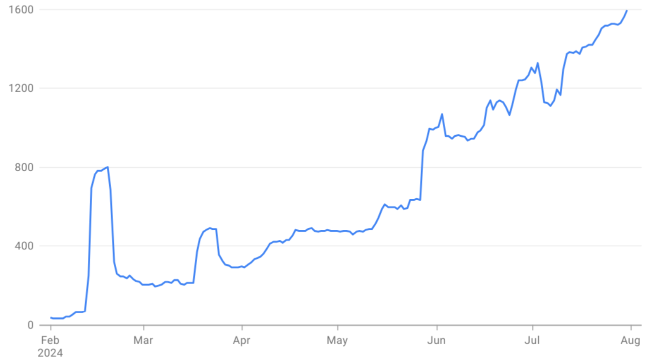

Figure 4: Seven-day-active users of pipe syntax in F1.

The third spike in June followed a SQL workshop presented at a user conference, to users with a wide range of SQL knowledge. The workshop included a 40-minute tutorial on pipe syntax, which was enough time to introduce and teach the language, walking through several examples applying it. Feedback showed workshop users were excited about pipe syntax, and we've seen significant continued usage.

Users are applying SQL with pipe syntax for widely varying use cases, including ad hoc queries, queries backing dashboards and reports, data processing pipelines, and libraries of reusable SQL functions and TVFs.

We've seen this adoption despite limited documentation and incomplete tooling (e.g. less auto-complete support inside pipe syntax). This adoption has been self-serve with minimal need for support. Many users understand pipe syntax immediately and find it compelling, reporting that it greatly improves their productivity and user experience in SQL. Several users have commented that they "love" using SQL with pipe syntax. The pain points solved here clearly resonate with users.

We haven't yet done formal user experience research on SQL with pipe syntax but plan to as future work. We conjecture that pipe syntax is easier to teach new users because it decouples SQL's conceptual ideas (aggregation, outer joins, window functions, etc) from syntactic complexity. Teaching pipe syntax first may even accelerate teaching standard syntax, since users can learn the relational operations first, understanding clearly how they work, and later learn how those operations are expressed (poorly) in standard SQL.

5.2 Complex queries

Pipe syntax naturally expresses queries with linear operator structure. Complex queries with tree-like structure don't map as directly to pipe syntax.

A common pattern is to start a query by joining data from several sources (in a `FROM` clause) and then apply a linear sequence of operations (filtering, aggregating, projecting, ordering, etc) to compute the result after those initial joins. This pattern works well using pipe operators for the work following the initial `FROM`.

[6] Operations like `INSERT INTO` and `CREATE [TEMP] TABLE` could also work as terminal pipe operators on the end of a query. We haven't tried that so far. Keeping those verbs at the start distinguishes pure queries from statements with side-effects.

Operator	Behavior and Usage
ASSERT <condition> [, <message>]	This adds assertions in SQL – much like assertions in other languages. Tables flow through unchanged, while validating <condition> for every row. If it's false for any row, the query fails, including the optional <message> in the error. Note: This acts as a partial optimization barrier, since reordering certain operations could lead to assertion failures.
LOG [(<pipe query fragment>)]	This logs part of an intermediate result table to a side channel – much like printf debugging in other languages. When debugging an unexpected query result, logging can be useful to examine what data existed earlier in the query. With no argument, this logs full rows from the table. A query fragment can be provided to narrow the output with filtering, aggregation, projections, etc.
DESCRIBE	This is used like a DESCRIBE table statement, but describes intermediate schema rather than a table. It returns one row containing a textual description of the input table schema, with column names and types, including hidden details like table aliases. This metadata can be helpful when writing queries or debugging why queries are invalid. This can run in constant time without physically executing the input query.
STATIC_DESCRIBE	This is like DESCRIBE but runs at analysis time, sending output to a logging side channel rather than as query output. It's a no-op at run time so it can be inserted anywhere to extract metadata without affecting query behavior. "Static" means it can produce log output without running the query, including in invalid queries (before the error), EXPLAIN, etc.

Figure 5: Experimental pipe operators which have no standard SQL equivalent

The FROM clause itself is most often written sequentially using JOIN ON/USING syntax, expressing a left-deep join tree. Left-deep joins are exactly what the pipe JOIN operator expresses, so a FROM clause written with JOIN syntax can be written with pipe JOINs easily. Pipe joins offer additional flexibility, since projections (SELECT or EXTEND), filtering (WHERE) or even aggregation can easily be done between joins, without any subqueries.

When the first item in a FROM clause is a table subquery, that can naturally be unnested as the pipe input query. Joins having subqueries on the right (non-left-deep joins) can still be expressed as a pipe JOIN to the same subquery.

Other techniques for factoring tree-like or DAG-like queries still apply. Subqueries can be extracted as WITH clauses or temp tables, where this helps to express complex structure or break up long queries.

In practice, breaking up long queries is somewhat less necessary, since pipe queries are shorter and don't need as much nesting or indentation.

5.3 Editing and debugging workflows

Pipe syntax works particularly well while editing or debugging SQL queries. Since logic flows top-to-bottom rather than inside-out, users can write queries sequentially, starting with a simple table scan and then incrementally adding operators, running the query as necessary to observe the result so far. Building queries incrementally in standard syntax is more difficult, often requiring adding wrapper subqueries, with new logic above and below the subquery.

Adding logic (filtering, projection, etc) anywhere within a query is easy, simply by inserting pipe operators, without refactoring or subqueries. It's surprisingly convenient that individual pipe operators can be trivially commented out.

For debugging, queries in pipe syntax have a useful **prefix property**. Each prefix of a query (up to a pipe character) is also a valid query. In editors with a "run selection" feature, users can select a prefix of the query and run it to see the intermediate query result up to that point. (Without editor support, adding a semicolon or commenting out the tail of the query also works.) For example, users can easily see the rows before and after an aggregation in the middle of a query by running two prefixes of the query.

During ad hoc debugging, it's also very convenient to add a LIMIT, WHERE or AGGREGATE on the end of a query, or after a selected query prefix, to see interesting rows or statistics on a final or intermediate result. For example, adding |> AGGREGATE COUNT(*) GROUP BY column extracts and counts the distinct values in some column.

Ad hoc data exploration often involves computing different aggregates of the same data. This is easy with pipe syntax with the combined AGGREGATE .. GROUP BY operator since alternate grouping can be computed by editing that single clause, without needing to update (or even include) a corresponding SELECT.

5.4 IDE and tooling support

SQL development environments are generally much less helpful than modern IDEs for other languages. Standard SQL is not amenable to typical IDE features because queries are too long and inconveniently structured, unlike imperative languages with short independent statements.

Auto-completion is difficult in standard SQL. The clauses are in the wrong order – auto-completing in SELECT isn't possible until after the FROM clause is written below to indicate what tables are in scope. Many changes require edits in multiple places (e.g. SELECT and GROUP BY, and repeatedly SELECTing the same column in subqueries), so adding completions at one cursor location is inadequate. Many edits require refactoring first, adding subqueries rather than just inserting text.

Auto-completion works well with pipe SQL. The query flows sequentially, top to bottom, so the required context is available from text above. New operators are often added at the end of the query, but adding new operators in the middle also works. The prefix property means the auto-completer can analyze a query prefix to know exactly which names are in scope at any point. Many changes are possible with one local edit, without requiring corresponding edits or refactoring elsewhere.

The prefix property makes building an interactive SQL debugger plausible. An IDE could allow clicking on any pipe operator and then displaying intermediate query results at that operator, with metadata about columns in scope, or stats about that result. Single-stepping through pipe operators could work, showing the intermediate result after each operator. (Single-stepping appears to imply imperative semantics, but the query prefix to that point could still be treated as a declarative query and optimized as usual. The user would step through business logic incrementally, not physical operators.) Debugging like this wouldn't work well with standard syntax because it's difficult to isolate substrings (not prefixes) of a standard syntax query as usefully executable fragments, other than whole subqueries. Combining tool support with language extensions (see 4.3.3) could be even more powerful.

5.5 SQL code generators and rewriters

SQL code generators are common in reporting tools, object-relational mappings (ORMs) and other contexts. Generating standard SQL is excessively difficult. An application typically has some data structure or API describing the desired query as logical operations, generally corresponding to relational operation. Converting these operations into standard SQL is difficult because SQL syntax doesn't express those operators naturally or composably. SQL generators typically generate subqueries for many operations since that's the easiest way to translate operations independently and reliably. The output queries are typically verbose and unreadable, with deeply nested subqueries and indirection through layers of aliasing.

Pipe syntax is easier to generate directly, since relational operations can be expressed with a single pipe operator and they can be stacked sequentially. Most wrapper subqueries and layers of aliasing aren't needed. (Subqueries are still useful for `JOIN` right-hand-sides and expression subqueries.) The resulting queries can be much shorter, and are structured more like how hand-written versions of the same query would be expressed.

The same advantages apply in applications that rewrite SQL queries. For example, Google has applications that rewrite user queries to add differential privacy or other policy controls. SQL rewriting is usually done by resolving a query to algebraic form, applying transformations, and then regenerating a new query. Rewritten queries then have no connection to the original query, and are not human-readable. With pipe syntax, most rewrites involve adding or modifying individual operators (e.g. adding filtering and adjusting aggregations), and it's theoretically practical to apply these rewrites correctly while still preserving the original query structure, with its original names, comments, etc. This is an area for future work.

5.6 Potential applications with AI

Applying AI, particularly large language models (LLMs), around SQL queries is an active research area, including query generation and query understanding. Standard SQL is a challenging language for LLMs[22], including for some of the same reasons SQL is difficult for human readers and writers. SQL has huge query statements with significant internal cross-referencing that don't break up naturally into smaller independent steps. This may be one reason SQL LLMs have been less successful so far than LLM assistants for other languages like C++ and Java.

Pipe syntax makes SQL more like typical imperative languages, with independent sequential operators, with clear state between operators, with less action-at-a-distance. Speculatively, an LLM that understands pipe SQL syntax could more clearly understand operations in a query, and should be able to generate queries more accurately from its "mental model" of desired query operations. Generating valid pipe SQL should be easier for LLMs, the same way it's easier for algorithmic query generators (and for humans). The same advantages apply for assistive SQL coding – since more edits can be applied locally, it's easier to make suggestions for useful local edits, making use of accurate metadata that says exactly what's in scope at any particular point.

Human validation of AI-generated queries in standard SQL is extremely difficult since generated SQL is so verbose and unreadable. Generating SQL with pipe syntax, or translating generated SQL to pipe syntax, can produce more concise and readable SQL, facilitating human evaluation and validation.

5.7 Translating SQL to pipe syntax

Automated translation of standard syntax to pipe syntax is useful for several reasons:

- To help users learn pipe syntax.
- To assist users migrating queries to pipe syntax.
- To facilitate query evolution. Editing SQL is easier after converting to pipe syntax.
- To make queries understandable. Legacy SQL codebases often include large and complex queries which are difficult to understand. Converting to pipe syntax should simplify queries and make them easier to understand, while also facilitating further refactoring and simplification.
- For debugging. Expert users often help others debug queries, for correctness or performance. The first step is often decoding a complex query to understand what it's doing. Translating to pipe syntax first can make the query shorter, simpler and easier to understand.
- To simplify generated queries. Generated queries converted to pipe syntax should be much easier to read.

Automated translation is an area for future work, but seems straightforward, applying a series of algorithmic refactoring steps, that reorder clauses and convert to pipe operators, extract and flatten table subqueries, simplify or drop redundant `SELECT`s, etc.

6 IMPLEMENTATION

GoogleSQL implements SQL as a reusable component, shared by several query engines across Google, including F1, BigQuery, Spanner, Procella and others. GoogleSQL implements SQL parsing and language analysis, fully checking and resolving language semantics and producing a resolved algebraic representation that query engines consume and convert to physical execution plans. GoogleSQL also provides function libraries and other components query engines reuse in their implementations, and provides a comprehensive suite of compliance tests to validate behavior is correct and consistent across query engines.

This shared language analysis component allowed us to implement pipe syntax once, and then enable it in multiple query engines. Inside Google, we did initial development with F1, with early users, and then expanded to other internal SQL tools. We're now exploring enabling pipe syntax for use in public BigQuery and Spanner products in Google Cloud.

Pipe syntax is also in the open source release, ZetaSQL[12].

Pipe syntax is implemented entirely within GoogleSQL parsing and analysis code, and required minimal work for query engines to enable. This analysis produces exactly the same algebra, with the same operations as standard SQL, so query engines simply enable the language analysis feature. No new execution or optimization features are needed. (Our pipe operators so far aim for parity with existing SQL. Novel operators may be added later.)

This approach makes pipe syntax immediately a first-class feature in query engines that enable it, making the syntax available and fully interoperable everywhere SQL queries are used.

Supporting pipe syntax via query translation using a tool or proxy is also possible. Our initial implementation used translation, implemented in an `Unpipefier` library that analyzes queries with pipe syntax enabled and then uses our `SQLBuilder` library to generate a standard-syntax equivalent supported by target systems.

While pipe syntax appears to add significant new language, with a well-structured implementation, the complexity is manageable. The new grammar mostly reuses fragments of existing SQL clauses, and analysis code similarly reuses existing logic. Most pipe operators were implemented by refactoring existing analysis code so logic could be shared across standard and pipe operators rather than creating parallel forks of similar logic. The underlying infrastructure for query resolving, like the type system, name scoping, and expression handling, are all reused as is.

Adding pipe syntax to a SQL implementation is drastically less work than implementing a new query language, particularly if trying to achieve parity with SQL functionality.

7 FUTURE WORK

Our initial implementation and work with enthusiastic SQL users has been helpful while refining the language, adding features and making adjustments to cover more query patterns and make the language easier to use. We are now expanding to all SQL users in Google, including potential users who might now choose to use SQL given the improved syntax. We're also exploring supporting pipe syntax externally in BigQuery and Spanner. Over time, we will collect more data on usage and patterns. Collecting statistics on average query length, nesting depth, etc, will be interesting.

Our goal with this paper is to share what we've learned so far. We believe the pipe syntax extensions are useful and would be appealing to most SQL users, and could potentially be adopted more widely across the industry. Future standardization of pipe SQL syntax could be worth exploring if there is broad interest. We believe pipe syntax offers a transformative improvement to SQL syntax that could be a good step forwards for the second 50 years of SQL.

Our initial work focused on achieving parity with the expressibility of standard SQL. There are many interesting possible features beyond parity, including features in other data products or APIs that don't yet exist in SQL. Adding features in standard SQL syntax is difficult, but adding extensions as new pipe operators is easy. (See 4.3.) Pipe syntax creates a better platform for future experimentation and innovation in query functionality and expressibility. Figure 5 shows some of our experimental extensions.

Outside the language, we continue to explore tooling improvements across the SQL ecosystem. Pipe syntax facilitates better tooling and IDE support for SQL, including auto-completion, automated refactoring, AI assistants, and interactive debuggers. Our goal is to make SQL a first-class language, with SQL data engineering tools and workflows that work as well as software engineering tools and workflows optimized for C++, Java, etc.

8 CONCLUSION

SQL has been extremely successful over its 50-year history, as the one standard language for all databases and query engines. That success has come despite serious language design issues that make SQL more difficult to learn and use than it needs to be.

SQL has been criticized repeatedly, and many replacements have been proposed. No replacement languages have achieved broad success or broad market penetration. Despite its flaws, SQL has a number of features that make it uniquely well-suited for declarative query processing, and the SQL ecosystem and userbase is too large to easily disrupt. It's exceedingly unlikely that SQL will be replaced by a successor language any time soon.

But we don't need to live with SQL's flaws. The language is fixable! This paper shows how pipe-structured data flow, inspired by other languages and APIs, can be added to SQL with moderate effort. **The resulting language is still SQL, but it's a better SQL.** It's more flexible, more extensible, and easier to use. Proficient SQL users can learn and adopt pipe syntax with minimal effort, getting immediate productivity gains and transforming how they work with SQL. These improvements go beyond incremental convenience and cosmetic improvements. Pipe syntax unlocks entirely new ways to work with SQL, opportunities for improved SQL tooling, and future language innovation.

Our experience so far suggests pipe syntax is a compelling improvement to SQL syntax that could be worth adoption across the industry. Replacing SQL isn't necessary, desirable or practical. We can fix SQL's most serious challenges from within the language.

9 ACKNOWLEDGEMENTS

We'd like to thank Goetz Graefe and John Cieslewicz for their support and feedback, and to our early users who gave feedback. This work is possible thanks to the work of many current and former GoogleSQL team members, contributors, and partner teams building query engines and other SQL tools.

REFERENCES

[1] [n.d.]. Apache Beam. Retrieved 2024-07-12 from https://beam.apache.org

[2] [n.d.]. Google Cloud BigQuery. Retrieved 2024-07-12 from https://cloud.google.com/bigquery

[3] [n.d.]. Graph Query Language GQL. Retrieved 2024-07-12 from https://www.gqlstandards.org/

[4] [n.d.]. GROUP BY ALL syntax in BigQuery. Retrieved 2024-07-12 from https://cloud.google.com/bigquery/docs/reference/standard-sql/query-syntax#group_by_all

[5] [n.d.]. KQL query language. Retrieved 2024-07-12 from https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query

[6] [n.d.]. LookML. Retrieved 2024-07-12 from https://cloud.google.com/looker/docs/what-is-lookml

[7] [n.d.]. Malloy. Retrieved 2024-07-12 from https://www.malloydata.dev/

[8] [n.d.]. Match-Recognize. Retrieved 2024-07-12 from https://modern-sql.com/feature/match_recognize

[9] [n.d.]. Pandas. Retrieved 2024-07-12 from https://pandas.pydata.org/docs/user_guide/index.html

[10] [n.d.]. Pipe Operator (|>) for JavaScript. Retrieved 2024-07-12 from https://github.com/tc39/proposal-pipeline-operator

[11] [n.d.]. PRQL. Retrieved 2024-07-12 from https://prql-lang.org/

[12] [n.d.]. ZetaSQL. Retrieved 2024-07-12 from https://github.com/google/zetasql

[13] Peter Bailis, Joseph M. Hellerstein, and Michael Stonebraker (Eds.). 2015. *Readings in Database Systems, 5th Edition.* http://www.redbook.io/

[14] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1757–1772. https://doi.org/10.1145/3299869.3314040

[15] Jamie Brandon. 2021. Against SQL. Retrieved 2024-07-12 from https://www.scattered-thoughts.net/writing/against-sql

[16] Don Chamberlin. 2023. 49 Years of Queries. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3555041.3589336

[17] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Ann Arbor, Michigan) *(SIGFIDET '74)*. Association for Computing Machinery, New York, NY, USA, 249–264. https://doi.org/10.1145/800296.811515

[18] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: unifying serving and analytical data at YouTube. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2022–2034. https://doi.org/10.14778/3352063.3352121

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 331–343. https://doi.org/10.1145/3035918.3056103

[20] Conor Cunningham, César A. Galindo-Legaria, and Goetz Graefe. 2004. PIVOT and UNPIVOT: optimization and execution strategies in an RDBMS. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 998–1009.

[21] C. J. Date. 1984. A critique of the SQL database language. *SIGMOD Rec.* 14, 3 (nov 1984), 8–54. https://doi.org/10.1145/984549.984551

[22] Avrilia Floratou et al. 2024. NL2SQL is a solved problem... Not! *(CIDR 2024)*.

[23] Janine Guertler. 2022. System Migration Risks: 7 Reasons Why Migrations Fail. Retrieved 2024-07-12 from https://www.celonis.com/blog/seven-reasons-why-system-migrations-fail/

[24] Alex Monahan. 2023. Even Friendlier SQL with DuckDB. Retrieved 2024-07-12 from https://duckdb.org/2023/08/23/even-friendlier-sql.html

[25] Thomas Neumann and Viktor Leis. 2024. A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language *(CIDR 2024)*.

[26] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR* abs/1405.3631 (2014). arXiv:1405.3631 http://arxiv.org/abs/1405.3631

[27] Serge Rielau. 2023. Databricks SQL Year in Review (Part II): SQL Programming Features. Retrieved 2024-07-12 from https://www.databricks.com/blog/databricks-sql-year-review-part-ii-sql-programming-features

[28] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. 2018. F1 query: declarative querying at scale. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1835–1848. https://doi.org/10.14778/3229863.3229871

[29] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *SIGMOD Record* 53, 2 (Jun 2024), 21–37. https://db.cs.cmu.edu/papers/2024/whatgoesaround-sigmodrec2024.pdf

[30] Herb Sutter. 2020. Bridge to NewThingia: How to answer "why will yours succeed, when others have failed?". Retrieved 2024-07-12 from https://www.youtube.com/watch?v=wIHfaH9Kffs

[31] Wen Zhang and Xi Cheng. 2023. Introducing BigQuery text embeddings for NLP tasks. Retrieved 2024-07-12 from https://cloud.google.com/blog/products/data-analytics/introducing-bigquery-text-embeddings