wvlet

Wvlet: A New Flow-Style Query Language

For Functional Data Modeling and Interactive Data Analysis

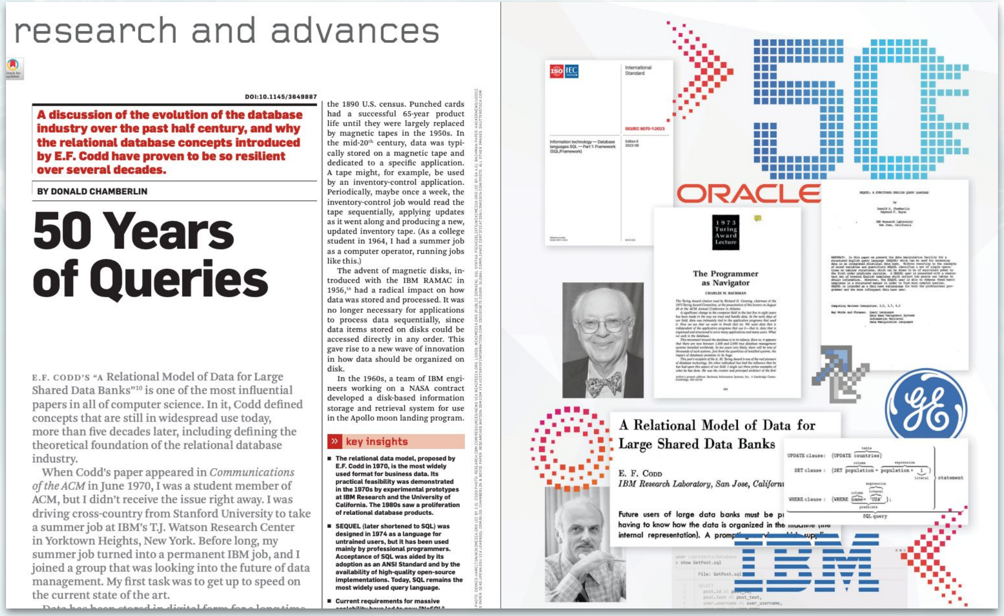
December 12, 2024
Trino Summit 2024

Taro L. Saito (Leo)
Senior Principal Engineer at Treasure Data

TREASURE DATA

2024: 50th Anniversary of SQL

- SQL was born in 1974
    - SQL's early design choices paved the way for standardization and commercial adoption
- Since then, applications of SQL have expanded significantly
    - OLAP: From RDBMS to Trino, Hive, Spark, etc.
    - Embedded DB: SQLite (OLTP) -> DuckDB (OLAP with Parquet/Iceberg support)
    - SQL's user base has grown beyond DBA to include non-engineers



research and advances

DOI:10.1145/3649887

A discussion of the evolution of the database industry over the past half century, and why the relational database concepts introduced by E.F. Codd have proven to be so resilient over several decades.

BY DONALD CHAMBERLIN

50 Years of Queries

E.F. CODD'S "A Relational Model of Data for Large Shared Data Banks"[10] is one of the most influential papers in all of computer science. In it, Codd defined concepts that are still in widespread use today, more than five decades later, including defining the theoretical foundation of the relational database industry.

When Codd's paper appeared in *Communications of the ACM* in June 1970, I was a student member of ACM, but I didn't receive the issue right away. I was driving cross-country from Stanford University to take a summer job at IBM's T.J. Watson Research Center in Yorktown Heights, New York. Before long, my summer job turned into a permanent IBM job, and I joined a group that was looking into the future of data management. My first task was to get up to speed on the current state of the art.

the 1890 U.S. census. Punched cards had a successful 65-year product life until they were largely replaced by magnetic tapes in the 1950s. In the mid-20th century, data was typically stored on a magnetic tape and dedicated to a specific application. A tape might, for example, be used by an inventory-control application. Periodically, maybe once a week, the inventory-control job would read the tape sequentially, applying updates as it went along and producing a new, updated inventory tape. (As a college student in 1964, I had a summer job as a computer operator, running jobs like this.)

The advent of magnetic disks, introduced with the IBM RAMAC in 1956,[16] had a radical impact on how data was stored and processed. It was no longer necessary for applications to process data sequentially, since data items stored on disks could be accessed directly in any order. This gave rise to a new wave of innovation in how data should be organized on disk.

In the 1960s, a team of IBM engineers working on a NASA contract developed a disk-based information storage and retrieval system for use in the Apollo moon landing program.

» key insights

- **The relational data model, proposed by E.F. Codd in 1970, is the most widely used format for business data. Its practical feasibility was demonstrated in the 1970s by experimental prototypes at IBM Research and the University of California. The 1980s saw a proliferation of relational database products.**
- **SEQUEL (later shortened to SQL) was designed in 1974 as a language for untrained users, but it has been used mainly by professional programmers. Acceptance of SQL was aided by its adoption as an ANSI Standard and by the availability of high-quality open-source implementations. Today, SQL remains the most widely used query language.**
- **Current requirements for massive**

50
ORACLE
IBM
International Standard
The Programmer as Navigator
1973 Turing Award Lecture
A Relational Model of Data for Large Shared Data Banks
E. F. Codd
IBM Research Laboratory, San Jose, California
UPDATE clause
SET clause
WHERE clause
SQL query

Trino at Treasure Data (2024)

- Treasure Data has been operating Trino (formerly Presto) as a service since 2014 (10th anniversary!)
  - 3+ million Trino SQL queries processed / day
  - 400+ trillion rows processed / day
  - 3+ billion S3 GET requests / day
    - Reduced from 10 billion requests / day (2023) by partition optimization
- In 2024, we completed the customer traffic migration from Presto (350) to Trino
  - See our migration/test methods in DBTest 2022 paper
- A lot of challenges in managing SQL and helping users (or LLM) write efficient SQL queries

Journey of Migrating Millions of Queries on The Cloud

Taro L. Saito, Naoki Takezoe, Yukihiro Okada, Takako Shimamoto
Dongmin Yu, Suprith Chandrashekharachar, Kai Sasaki, Shohei Okumiya
Yan Wang, Takashi Kurihara, Ryu Kobayashi, Keisuke Suzuki
Zhenghong Yang, Makoto Onizuka*
Treasure Data
Mountain View, CA, USA

[cs.DB] 17 May 2022

ABSTRACT

Treasure Data is processing millions of distributed SQL queries every day on the cloud. Upgrading the query engine service at this scale is challenging because we need to migrate all of the production queries of the customers to a new version while preserving the correctness and performance of the data processing pipelines. To ensure the quality of the query engines, we utilize our query logs to build customer-specific benchmarks and replay these queries with real customer data in a secure pre-production environment. To simulate millions of queries, we need effective minimization of test query sets and better reporting of the simulation results to proactively find incompatible changes and performance regression of the new version. This paper describes the overall design of our system and shares various challenges in maintaining the quality of the query engine service on the cloud.

ACM Reference Format:
Taro L. Saito, Naoki Takezoe, Yukihiro Okada, Takako Shimamoto, Dongmin

As of February 2022, Treasure Data is processing more than 1.5 million SQL queries every day coming from 5,000+ users in various regions, including US, EU, Japan, Korea, etc. The number of processed records exceeds 100 trillion rows/day, which is equivalent to processing 1.2 billion rows/sec. Our job as a service provider is preserving the behavior of these SQL queries while maintaining the underlying query engine versions up-to-date so that our customers can keep processing their data analysis pipelines without worrying about maintaining the data platform.

Maintaining A Data Platform on the Cloud. Treasure Data provides query engine services on the cloud by integrating open-source (OSS) distributed SQL query engines Trino (formerly known as Presto) [2] and Hive [1] on top of Amazon Web Service (AWS). Our target customers (e.g., marketers) are not dedicated system engineers, so they have a strong demand to have a data platform which is easy to use and has no need to maintain by themselves.

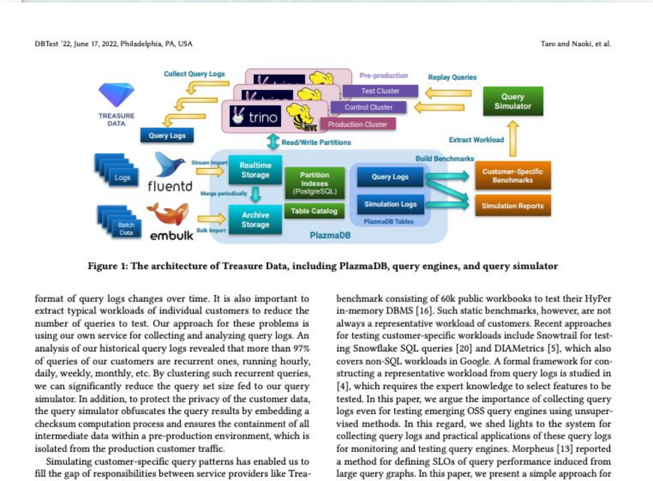


Figure 1: The architecture of Treasure Data, including PlazmaDB, query engines, and query simulator

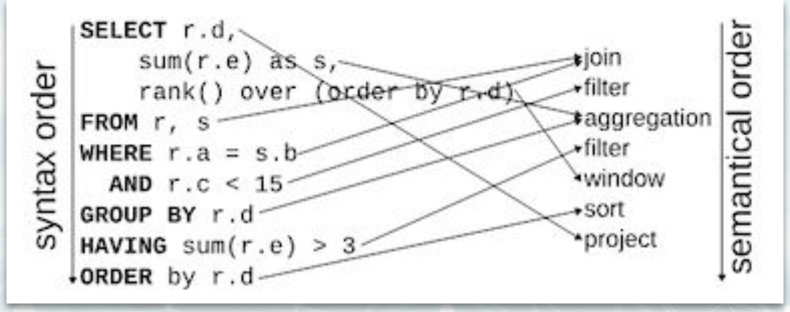
format of query logs changes over time. It is also important to extract typical workloads of individual customers to reduce the number of queries to test. Our approach for these problems is using our own service for collecting and analyzing query logs. An analysis of our historical query logs revealed that more than 97% of queries of our customers are recurrent ones, running hourly, daily, weekly, monthly, etc. By clustering such recurrent queries, we can significantly reduce the query set size fed to our query simulator. In addition, to protect the privacy of the customer data, the query simulator obfuscates the query results by embedding a checksum computation process and ensures the containment of all intermediate data within a pre-production environment, which is isolated from the production customer traffic.

Simulating customer-specific query patterns has enabled us to fill the gap of responsibilities between service providers like Trea-

benchmark consisting of 60k public workbooks to test their HyPer in-memory DBMS [16]. Such static benchmarks, however, are not always a representative workload of customers. Recent approaches for testing customer-specific workloads include Snowtrail for testing Snowflake SQL queries [20] and DIAMetrics [5], which also covers non-SQL workloads in Google. A formal framework for constructing a representative workload from query logs is studied in [4], which requires the expert knowledge to select features to be tested. In this paper, we argue the importance of collecting query logs even for testing emerging OSS query engines using unsupervised methods. In this regard, we shed lights to the system for collecting query logs and practical applications of these query logs for monitoring and testing query engines. Morpheus [13] reported a method for defining SLOs of query performance induced from large query graphs. In this paper, we present a simple approach for

What's Wrong with SQL?

- The syntactic order of SQL doesn't match the actual data flow
    - A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language (CIDR 24)
    - Even SQL experts find it challenging to debug nested queries

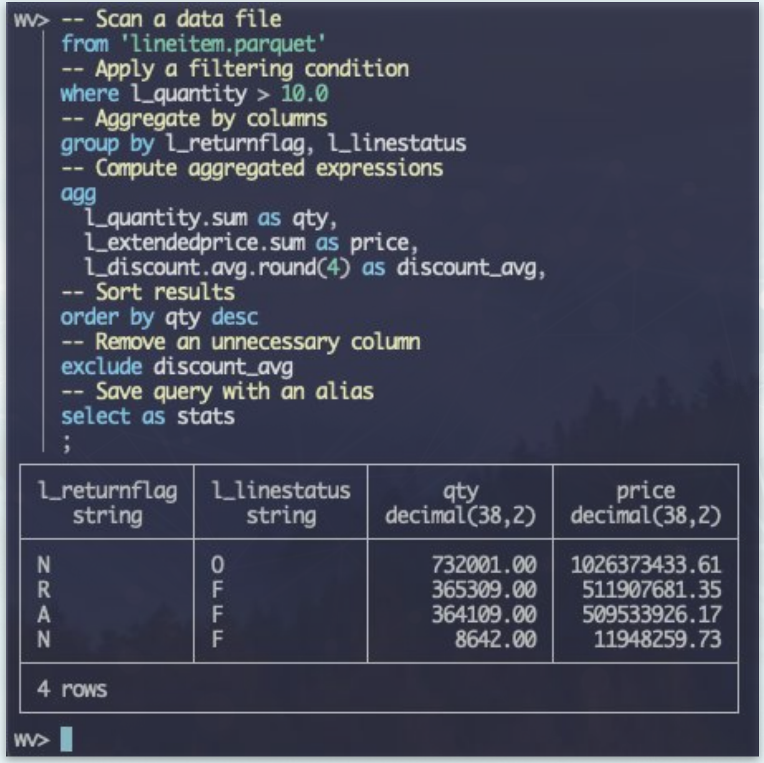


```
SELECT r.d,
    sum(r.e) as s,
    rank() over (order by r.d)
FROM r, s
WHERE r.a = s.b
  AND r.c < 15
GROUP BY r.d
HAVING sum(r.e) > 3
ORDER by r.d
```

- Lacks essential software engineering features for managing many queries
    - No built-in support for reusing queries
    - No entry points for multi-query optimization
        - e.g., incremental processing and pipeline execution like dbt
    - No built-in debugging or testing capabilities

Wvlet: Modernizing 50-Year-Old SQL

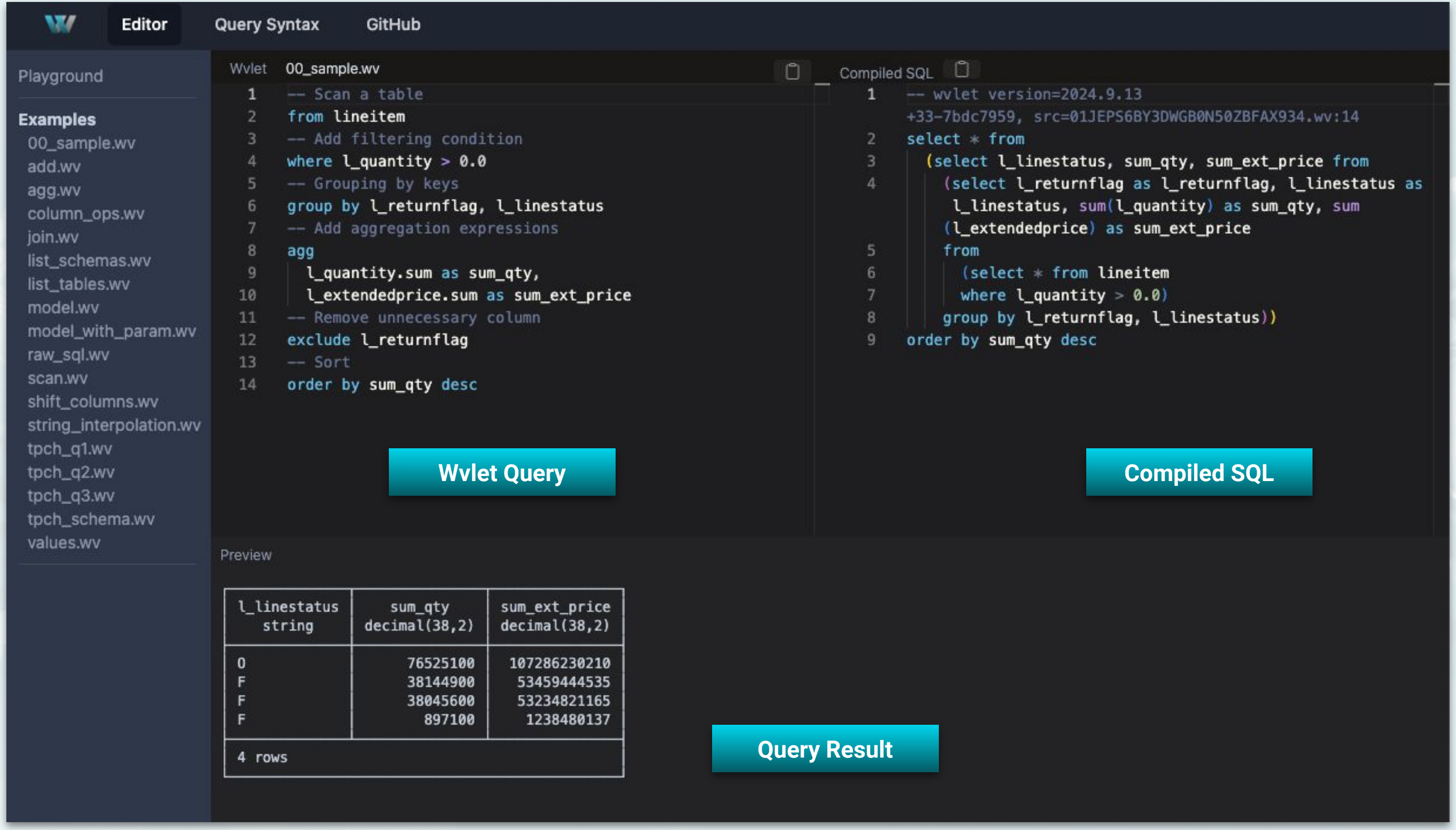
- SQL has proven to be effective and useful
  - Its natural-language based syntax has gained widespread adoption
- Redesigned SQL to be more intuitive and functional
- Wvlet: a new flow-style query language for weaving data
  - Pronounced as weave-let
  - Queries start with “from” for intuitive data flow
- Functional
  - Wvlet queries are reusable and composable like functions
- Cross SQL engine support
  - Generates SQL for Trino, Hive, DuckDB, etc.
- Try Wvlet on your web browser https://wvlet.org/
  - No installation is required



```
wv> -- Scan a data file
  │ from 'lineitem.parquet'
  │ -- Apply a filtering condition
  │ where l_quantity > 10.0
  │ -- Aggregate by columns
  │ group by l_returnflag, l_linestatus
  │ -- Compute aggregated expressions
  │ agg
  │   l_quantity.sum as qty,
  │   l_extendedprice.sum as price,
  │   l_discount.avg.round(4) as discount_avg,
  │ -- Sort results
  │ order by qty desc
  │ -- Remove an unnecessary column
  │ exclude discount_avg
  │ -- Save query with an alias
  │ select as stats
  │ ;
┌─────────────┬─────────────┬──────────────┬───────────────┐
│ l_returnflag│ l_linestatus│      qty     │     price     │
│    string   │    string   │ decimal(38,2)│ decimal(38,2) │
├─────────────┼─────────────┼──────────────┼───────────────┤
│ N           │ O           │     732001.00│  1026373433.61│
│ R           │ F           │     365309.00│   511907681.35│
│ A           │ F           │     364109.00│   509533926.17│
│ N           │ F           │       8642.00│    11948259.73│
├─────────────┴─────────────┴──────────────┴───────────────┤
│ 4 rows                                                   │
└──────────────────────────────────────────────────────────┘
wv>
```

Command-line editor of Wvlet (wv)

Wvlet Playground wvlet.org



```
-- Scan a table
from lineitem
-- Add filtering condition
where l_quantity > 0.0
-- Grouping by keys
group by l_returnflag, l_linestatus
-- Add aggregation expressions
agg
  l_quantity.sum as sum_qty,
  l_extendedprice.sum as sum_ext_price
-- Remove unnecessary column
exclude l_returnflag
-- Sort
order by sum_qty desc
```

```
-- wvlet version=2024.9.13+33-7bdc7959, src=01JEPS6BY3DWGB0N50ZBFAX934.wv:14
select * from
  (select l_linestatus, sum_qty, sum_ext_price from
    (select l_returnflag as l_returnflag, l_linestatus as l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_ext_price
    from
      (select * from lineitem
      where l_quantity > 0.0)
    group by l_returnflag, l_linestatus))
order by sum_qty desc
```

```
┌──────────────┬────────────────┬─────────────────┐
│ l_linestatus │    sum_qty     │  sum_ext_price  │
│    string    │ decimal(38,2)  │ decimal(38,2)   │
├──────────────┼────────────────┼─────────────────┤
│ O            │       76525100 │    107286230210 │
│ F            │       38144900 │     53459444535 │
│ F            │       38045600 │     53234821165 │
│ F            │         897100 │      1238480137 │
├──────────────┴────────────────┴─────────────────┤
│ 4 rows                                          │
└─────────────────────────────────────────────────┘
```

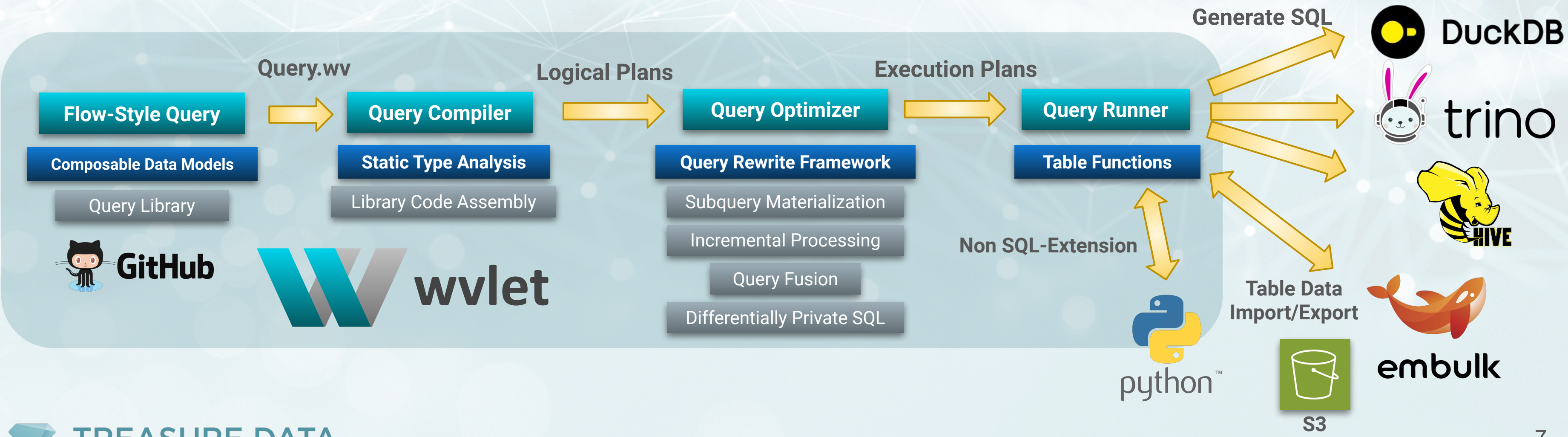
Wvlet Query

Compiled SQL

Query Result

Wvlet: Architecture

- From query (.wv) files, Wvlet compiler produces logical plans, execution plans, and SQL statements
- Logical Plans
  - Tree-representations of relational operators (e.g., scan, filter, projection)
    - Sort(Join(Filter(TableScan(...)), TableScan(...)))
- Execution Plans
  - A sequence of steps to execute SQL and other programs
    - ExecutePlan(ExecuteSQL(query), ExecuteTest(expr), ExecuteCommand(expr), …)



Flow-Style Query
Composable Data Models
Query Library
GitHub

Query.wv

Query Compiler
Static Type Analysis
Library Code Assembly
wvlet

Logical Plans

Query Optimizer
Query Rewrite Framework
Subquery Materialization
Incremental Processing
Query Fusion
Differentially Private SQL

Execution Plans

Query Runner
Table Functions

Non SQL-Extension
python

Generate SQL
DuckDB
trino
HIVE

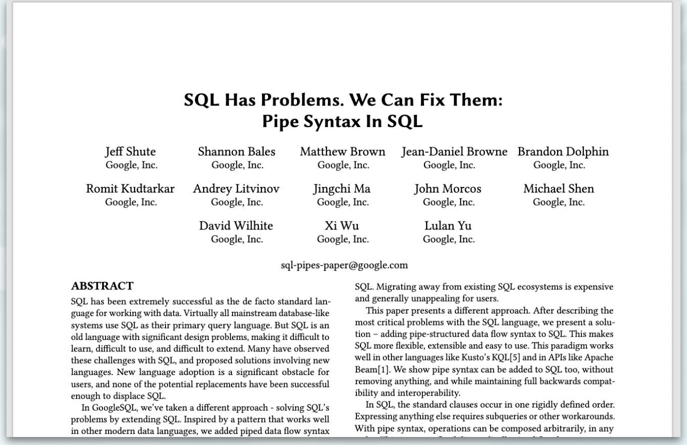
Table Data Import/Export
embulk
S3

Wvlet: Analyze As You Write

- Flow-style queries
  - Each line is a single operation
- Peek the data and schema at any point
  - A subquery becomes a range of lines in the query text
- Easy to extend
  - Queries can be reused for further analysis

Related Work: GoogleSQL Pipe Syntax (2024)

- SQL Has Problems. We Can Fix Them: Pipe Syntax In SQL (VLDB 2024)
- Extended SQL syntax with pipe operator (|>)
    - Available in ZetaSQL, an open-source SQL parser, used in Google products, e.g., BigQuery, F1, etc.
- Other flow-style languages:
    - PRQL, Microsoft Kusto, DryadLINQ, etc.



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

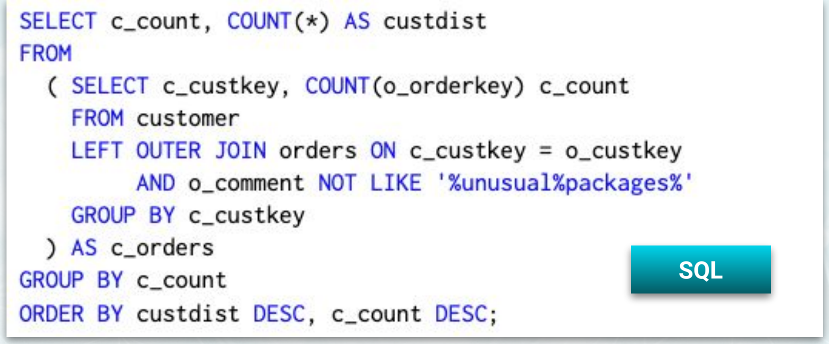
ABSTRACT

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 SQL. Migrating away from existing SQL ecosystems is expensive and generally unappealing for users.

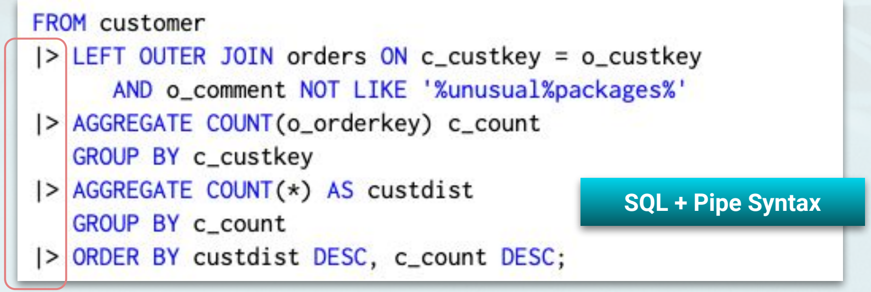
This paper presents a different approach. After describing the most critical problems with the SQL language, we present a solution – adding pipe-structured data flow syntax to SQL. This makes SQL more flexible, extensible and easy to use. This paradigm works well in other languages like Kusto's KQL[5] 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



```
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;
```

SQL



```
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;
```

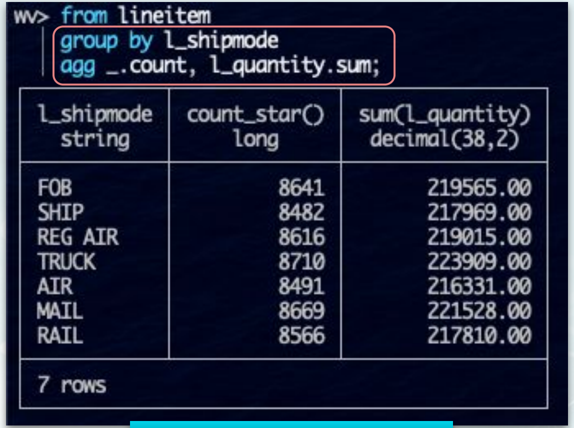
SQL + Pipe Syntax

Wvlet: Relational Operators (online reference)

- In Wvlet, all keywords must be lower-case letters
  - For consistency and reducing implementation efforts

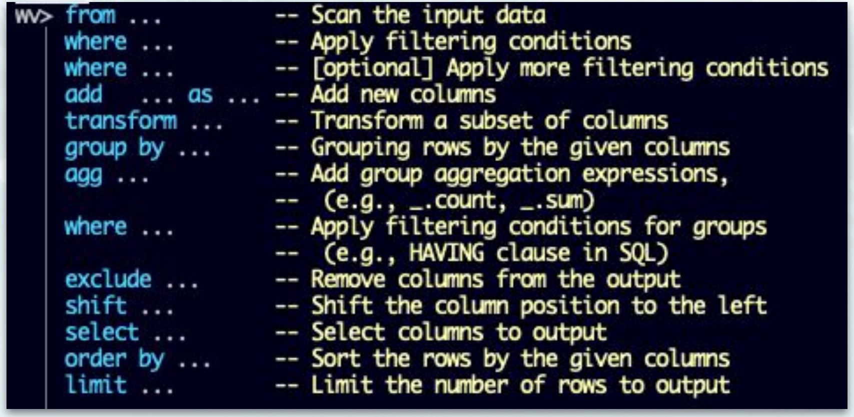
- Same operators with SQL:
  - **from, select**, **join**, **where**, **order by**, **limit**, etc.

- Aggregation in Wvlet
  - **group by** k1, k2, … **agg** …
  - **agg** (aggregation expr), **..**
    - Report group-by keys and aggregation expressions: k1, k2, …, expr1, expr2, …
  - **group by** k1, k2, … **where** …
    - equivalent to group by … having …



```
wv> from lineitem
  group by l_shipmode
  agg _.count, l_quantity.sum;
┌────────────┬──────────────┬─────────────────┐
│ l_shipmode │ count_star() │ sum(l_quantity) │
│   string   │     long     │  decimal(38,2)  │
├────────────┼──────────────┼─────────────────┤
│ FOB        │         8641 │       219565.00 │
│ SHIP       │         8482 │       217969.00 │
│ REG AIR    │         8616 │       219015.00 │
│ TRUCK      │         8710 │       223909.00 │
│ AIR        │         8491 │       216331.00 │
│ MAIL       │         8669 │       221528.00 │
│ RAIL       │         8566 │       217810.00 │
├────────────┴──────────────┴─────────────────┤
│ 7 rows                                      │
└─────────────────────────────────────────────┘
```

Aggregation Query



```
wv> from ...          -- Scan the input data
  │ where ...         -- Apply filtering conditions
  │ where ...         -- [optional] Apply more filtering conditions
  │ add    ... as ... -- Add new columns
  │ transform ...     -- Transform a subset of columns
  │ group by ...      -- Grouping rows by the given columns
  │ agg ...           -- Add group aggregation expressions,
  │                   --  (e.g., _.count, _.sum)
  │ where ...         -- Apply filtering conditions for groups
  │                   --  (e.g., HAVING clause in SQL)
  │ exclude ...       -- Remove columns from the output
  │ shift ...         -- Shift the column position to the left
  │ select ...        -- Select columns to output
  │ order by ...      -- Sort the rows by the given columns
  │ limit ...         -- Limit the number of rows to output
```

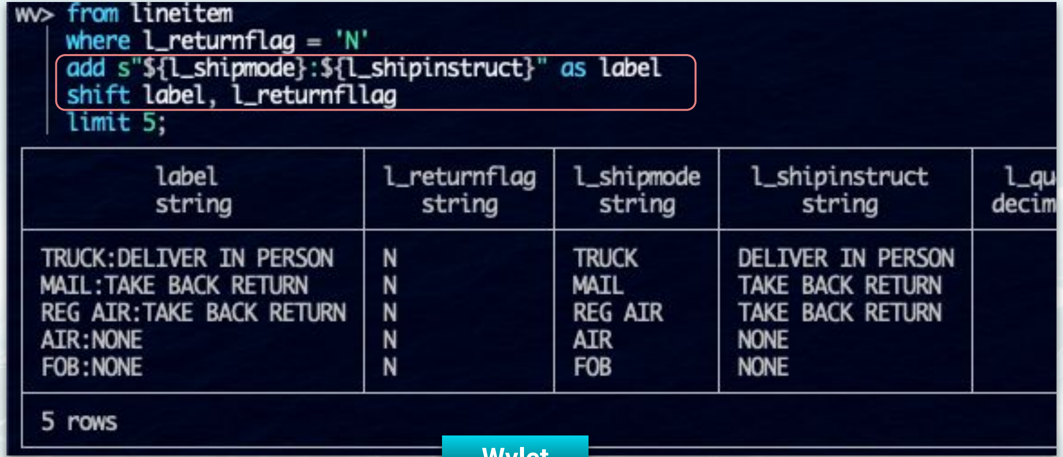
Basic Flow of Wvlet Queries

Wvlet: Column At A Time Manipulation

- For minimizing your typing effort

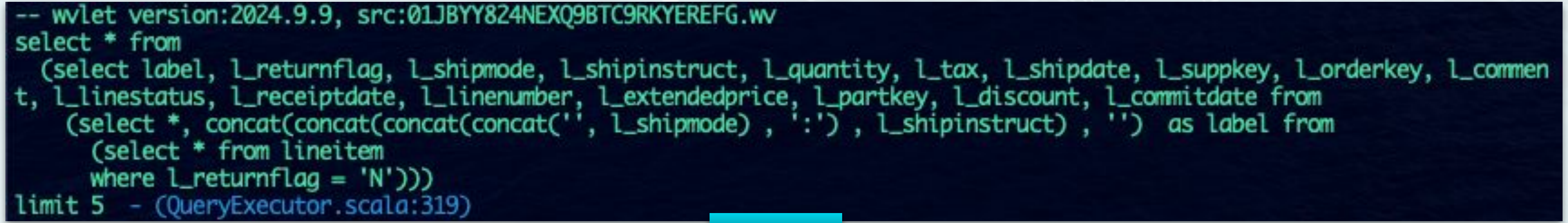
- **add** (expr) **as** …
  - Add a new column
- **rename** (column) as (new name)
- **exclude** (column), …
- **shift** (to left/right)? column, ….
  - Reorder columns for readability

- Changing a single column or its order in SQL is not easy as we need to enumerate all columns



```
wv> from lineitem
  | where l_returnflag = 'N'
  | add s"${l_shipmode}:${l_shipinstruct}" as label
  | shift label, l_returnfllag
  | limit 5;
┌────────────────────────────┬──────────────┬─────────────┬───────────────────┬──────
│           label            │ l_returnflag │ l_shipmode  │  l_shipinstruct   │ l_qu
│           string           │    string    │   string    │      string       │ decim
├────────────────────────────┼──────────────┼─────────────┼───────────────────┼──────
│ TRUCK:DELIVER IN PERSON    │ N            │ TRUCK       │ DELIVER IN PERSON │
│ MAIL:TAKE BACK RETURN      │ N            │ MAIL        │ TAKE BACK RETURN  │
│ REG AIR:TAKE BACK RETURN   │ N            │ REG AIR     │ TAKE BACK RETURN  │
│ AIR:NONE                   │ N            │ AIR         │ NONE              │
│ FOB:NONE                   │ N            │ FOB         │ NONE              │
├────────────────────────────┴──────────────┴─────────────┴───────────────────┴──────
│ 5 rows
└─────────────────────────────────────────────────────────────────────────────────────
```

Wvlet



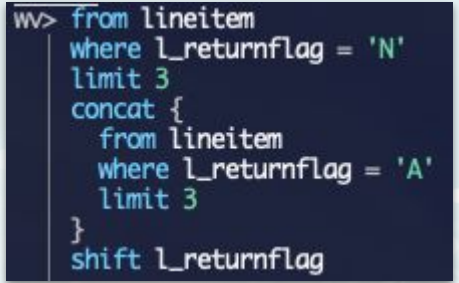
```
-- wvlet version:2024.9.9, src:01JBYY824NEXQ9BTC9RKYEREFG.wv
select * from
  (select label, l_returnflag, l_shipmode, l_shipinstruct, l_quantity, l_tax, l_shipdate, l_suppkey, l_orderkey, l_commen
t, l_linestatus, l_receiptdate, l_linenumber, l_extendedprice, l_partkey, l_discount, l_commitdate from
    (select *, concat(concat(concat(concat('', l_shipmode) , ':') , l_shipinstruct) , '')  as label from
      (select * from lineitem
      where l_returnflag = 'N')))
limit 5  - (QueryExecutor.scala:319)
```

SQL

Wvlet: Additional Relational Operators

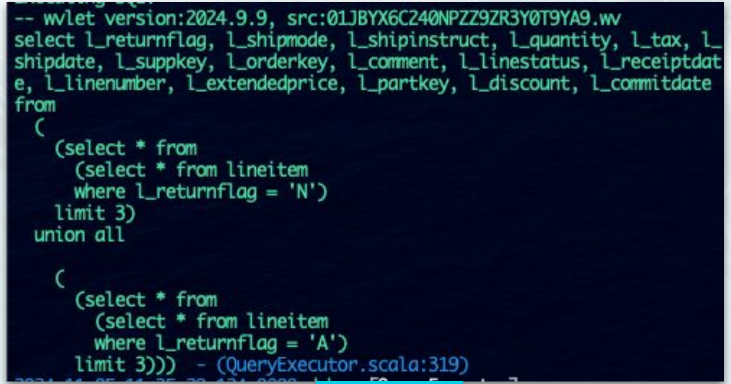
- Refined from SQL
  - **concat** = UNION ALL in SQL
    - Most of SQL users **wrongly use UNION (duplicate elimination)** where UNION ALL (concatenate two relations) is appropriate.
  - **dedup** = select distinct * (= duplicate elimination)

- Utility operators
  - **transform**: Update only subset of columns
  - **sample** *n*
  - **pivot**
    - Transform column values into individual columns
    - trino#1206 (Remains open since 2019)
    - Wvlet can run multiple SQL queries, so it's relatively easy to implement static/dynamic pivot



```
wv> from lineitem
    where l_returnflag = 'N'
    limit 3
    concat {
      from lineitem
      where l_returnflag = 'A'
      limit 3
    }
    shift l_returnflag
```

Wvlet

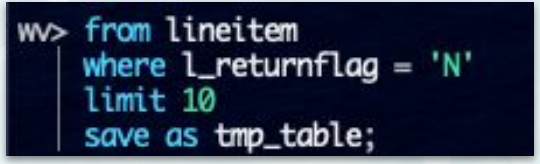


```
-- wvlet version:2024.9.9, src:01JBYX6C240NPZZ9ZR3Y0T9YA9.wv
select l_returnflag, l_shipmode, l_shipinstruct, l_quantity, l_tax, l_
shipdate, l_suppkey, l_orderkey, l_comment, l_linestatus, l_receiptdat
e, l_linenumber, l_extendedprice, l_partkey, l_discount, l_commitdate
from
  (
    (select * from
      (select * from lineitem
      where l_returnflag = 'N')
    limit 3)
  union all

    (
      (select * from
        (select * from lineitem
        where l_returnflag = 'A')
      limit 3))) - (QueryExecutor.scala:319)
```

SQL

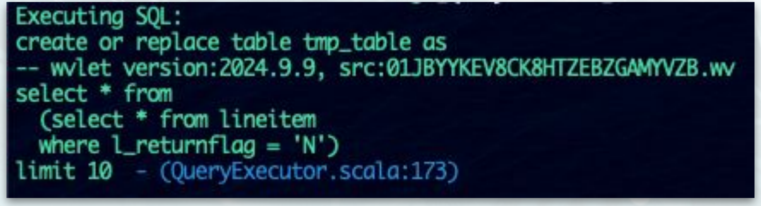
Wvlet: Update Statements

- **save as**
    - A shorthand notation for:
        - DROP TABLE IF EXISTS tbl; CREATE TABLE tbl AS SELECT …
            - Most frequently used pattern in Treasure Data
        - CREATE OR REPLACE TABLE AS SELECT
            - Available in DuckDB, Trino 431 (Since Oct 2023)
    - For DuckDB backend, you can save query results as Parquet/JSON files
- **append to**
    - Almost same with INSERT INTO, but it clarifies append-only semantics (no overwrites)
        - Useful when using versioned tables like Iceberg/Delta Lake
- **delete**
    - Delete selected rows from the table



```
wv> from lineitem
  | where l_returnflag = 'N'
  | limit 10
  | save as tmp_table;
```

Wvlet



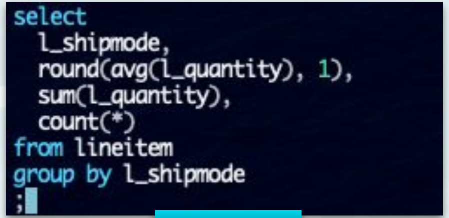
```
Executing SQL:
create or replace table tmp_table as
-- wvlet version:2024.9.9, src:01JBYYKEV8CK8HTZEBZGAMYVZB.wv
select * from
  (select * from lineitem
  where l_returnflag = 'N')
limit 10  - (QueryExecutor.scala:173)
```

SQL

Function Chaining via Dot Operator

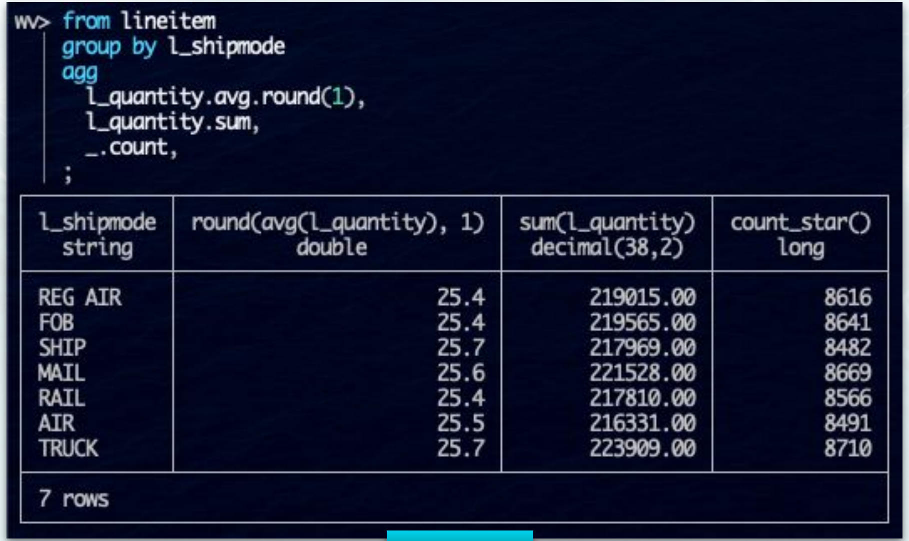
- In SQL, every function is global
    - SUM(x), AVG(x), COUNT(*), ROUND(x, 1), …
    - A lot of cursor movements are necessary to apply functions in SQL

- Wvlet supports chaining functions with dot operator as in modern programming languages:
    - e.g., x.sum, x.avg.round(2), _.count
    - _ (underscore) refers to the output from the previous operator



```
select
  l_shipmode,
  round(avg(l_quantity), 1),
  sum(l_quantity),
  count(*)
from lineitem
group by l_shipmode
;
```

SQL

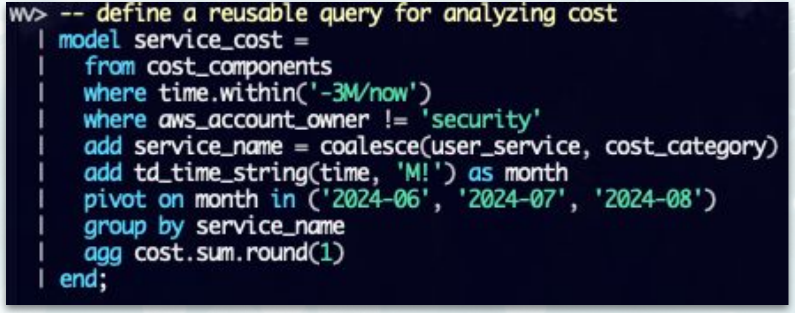


```
wv> from lineitem
  │ group by l_shipmode
  │ agg
  │   l_quantity.avg.round(1),
  │   l_quantity.sum,
  │   _.count,
  │ ;
┌────────────┬───────────────────────────┬─────────────────┬──────────────┐
│ l_shipmode │ round(avg(l_quantity), 1) │ sum(l_quantity) │ count_star() │
│   string   │          double           │ decimal(38,2)   │     long     │
├────────────┼───────────────────────────┼─────────────────┼──────────────┤
│ REG AIR    │                      25.4 │       219015.00 │         8616 │
│ FOB        │                      25.4 │       219565.00 │         8641 │
│ SHIP       │                      25.7 │       217969.00 │         8482 │
│ MAIL       │                      25.6 │       221528.00 │         8669 │
│ RAIL       │                      25.4 │       217810.00 │         8566 │
│ AIR        │                      25.5 │       216331.00 │         8491 │
│ TRUCK      │                      25.7 │       223909.00 │         8710 │
├────────────┴───────────────────────────┴─────────────────┴──────────────┤
│ 7 rows                                                                  │
└─────────────────────────────────────────────────────────────────────────┘
```

Wvlet

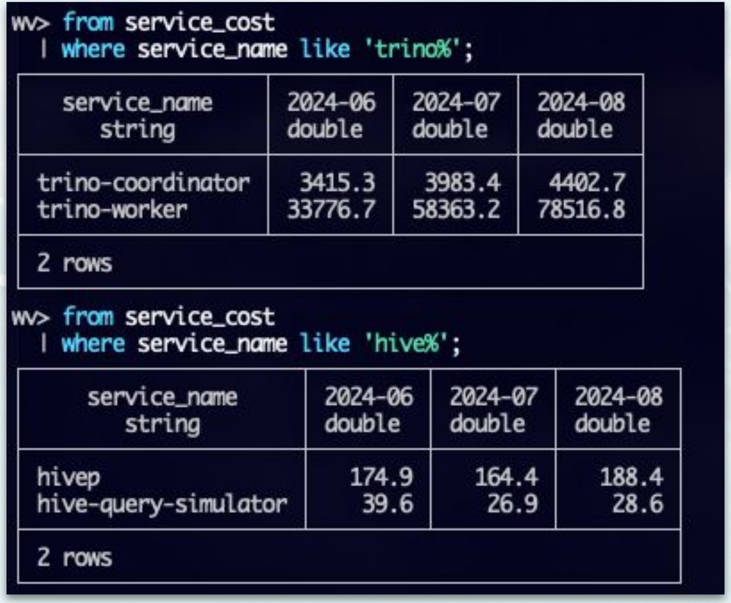
Wvlet Query = A Function for Building A Data Model

- In Wvlet, a query defined as a data model that can be reused at ease (Query Templates)
- Composable
  - You can add more relational operators (e.g., where, join) to data models
- Reusable
  - Models accept user parameters (= function arguments)



```
wv> -- define a reusable query for analyzing cost
  | model service_cost =
  |   from cost_components
  |   where time.within('-3M/now')
  |   where aws_account_owner != 'security'
  |   add service_name = coalesce(user_service, cost_category)
  |   add td_time_string(time, 'M!') as month
  |   pivot on month in ('2024-06', '2024-07', '2024-08')
  |   group by service_name
  |   agg cost.sum.round(1)
  | end;
```

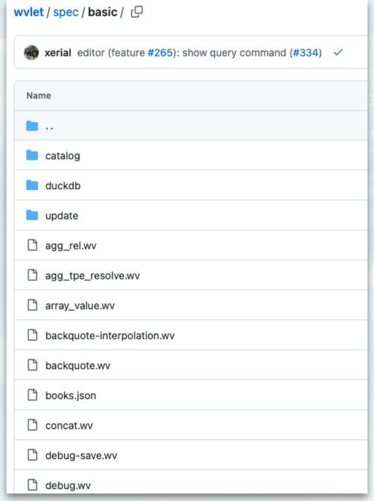
Reuse Queries



```
wv> from service_cost
  | where service_name like 'trino%';
┌───────────────────┬─────────┬─────────┬─────────┐
│   service_name    │ 2024-06 │ 2024-07 │ 2024-08 │
│      string       │ double  │ double  │ double  │
├───────────────────┼─────────┼─────────┼─────────┤
│ trino-coordinator │  3415.3 │  3983.4 │  4402.7 │
│ trino-worker      │ 33776.7 │ 58363.2 │ 78516.8 │
├───────────────────┴─────────┴─────────┴─────────┤
│ 2 rows                                          │
└─────────────────────────────────────────────────┘
wv> from service_cost
  | where service_name like 'hive%';
┌──────────────────────┬─────────┬─────────┬─────────┐
│     service_name     │ 2024-06 │ 2024-07 │ 2024-08 │
│        string        │ double  │ double  │ double  │
├──────────────────────┼─────────┼─────────┼─────────┤
│ hivep                │   174.9 │   164.4 │   188.4 │
│ hive-query-simulator │    39.6 │    26.9 │    28.6 │
├──────────────────────┴─────────┴─────────┴─────────┤
│ 2 rows                                             │
└────────────────────────────────────────────────────┘
```

Testing Queries

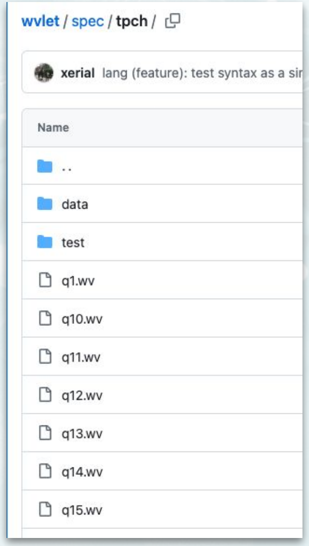
- Wvlet Test Syntax
    - Useful for verifying resulting schema and results
- Wvlet is tested with Wvlet
    - Wvlet spec queries
    - Covers all 22 TPC-H queries



wvlet / spec / basic /

xerial editor (feature #265): show query command (#334) ✓

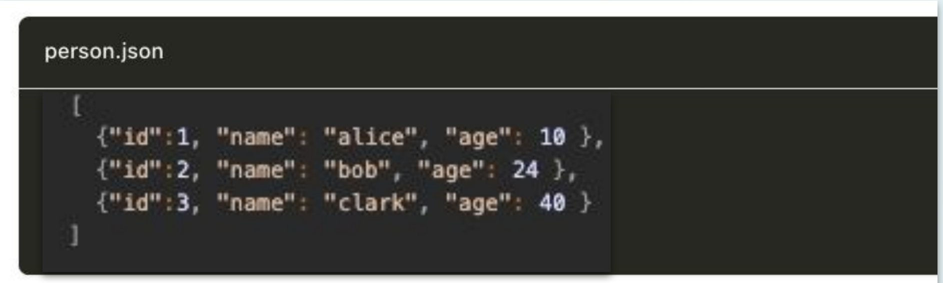
Name
..
catalog
duckdb
update
agg_rel.wv
agg_tpe_resolve.wv
array_value.wv
backquote-interpolation.wv
backquote.wv
books.json
concat.wv
debug-save.wv
debug.wv



wvlet / spec / tpch /

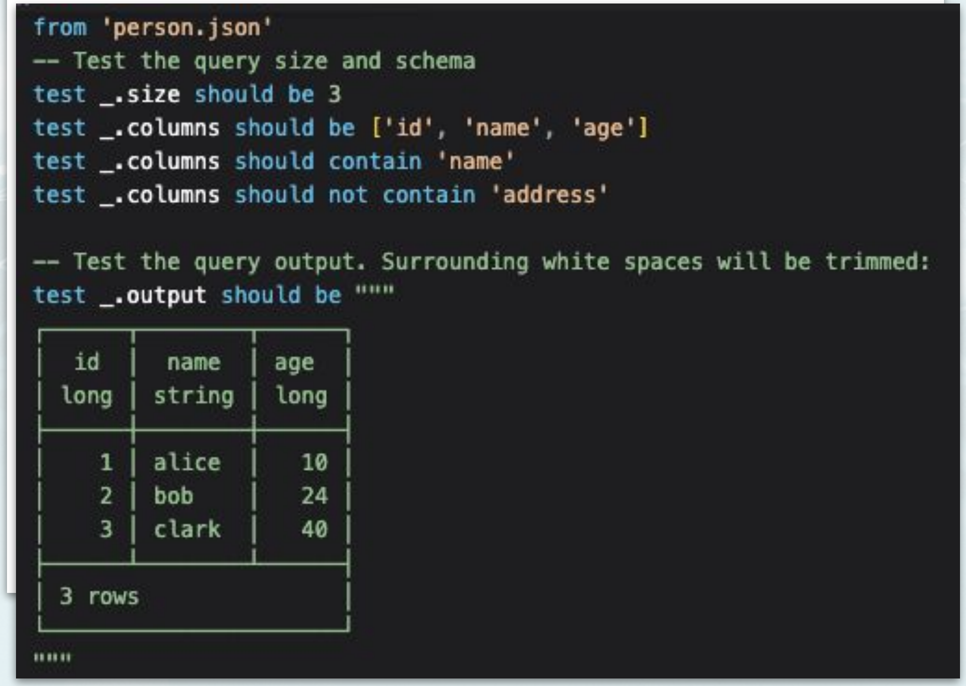
xerial lang (feature): test syntax as a sir

Name
..
data
test
q1.wv
q10.wv
q11.wv
q12.wv
q13.wv
q14.wv
q15.wv



person.json

```json
[
  {"id":1, "name": "alice", "age": 10 },
  {"id":2, "name": "bob", "age": 24 },
  {"id":3, "name": "clark", "age": 40 }
]
```

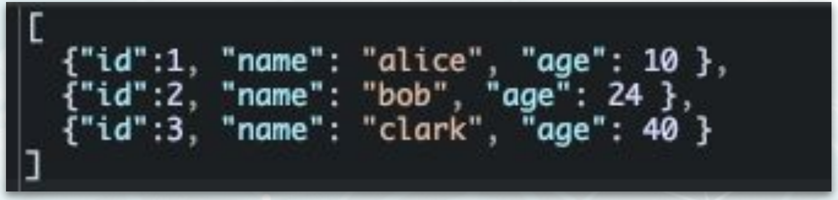


```
from 'person.json'
-- Test the query size and schema
test _.size should be 3
test _.columns should be ['id', 'name', 'age']
test _.columns should contain 'name'
test _.columns should not contain 'address'

-- Test the query output. Surrounding white spaces will be trimmed:
test _.output should be """
┌──────┬────────┬──────┐
│  id  │  name  │ age  │
│ long │ string │ long │
├──────┼────────┼──────┤
│    1 │ alice  │   10 │
│    2 │ bob    │   24 │
│    3 │ clark  │   40 │
├──────┴────────┴──────┤
│ 3 rows               │
└──────────────────────┘
"""
```

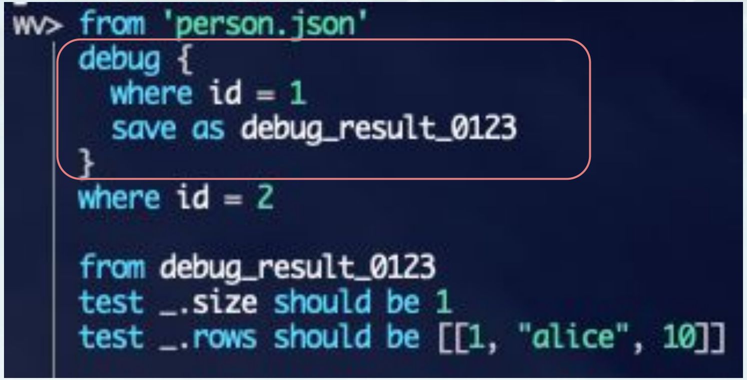
TREASURE DATA

Debugging Queries

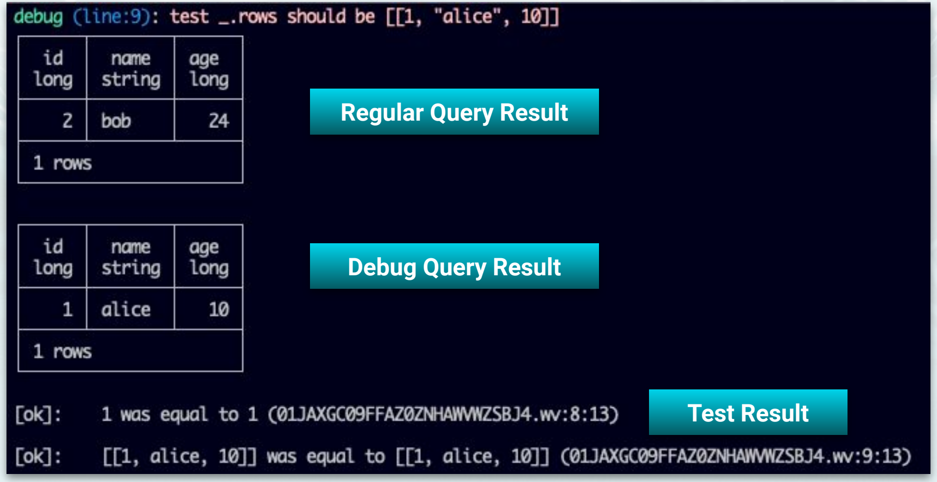
- Debug operator can be used for checking intermediate query results
- ExecutionPlanner generates multiple execution paths for debug and regular query evaluation
  - Debug path: Query before debug statement -> debug query
  - Regular path: Query without debug statement



```
[
  {"id":1, "name": "alice", "age": 10 },
  {"id":2, "name": "bob", "age": 24 },
  {"id":3, "name": "clark", "age": 40 }
]
```



```
wv> from 'person.json'
    debug {
      where id = 1
      save as debug_result_0123
    }
    where id = 2

    from debug_result_0123
    test _.size should be 1
    test _.rows should be [[1, "alice", 10]]
```



```
debug (line:9): test _.rows should be [[1, "alice", 10]]
┌──────┬────────┬──────┐
│ id   │ name   │ age  │
│ long │ string │ long │
├──────┼────────┼──────┤
│    2 │ bob    │   24 │
├──────┴────────┴──────┤
│ 1 rows               │
└──────────────────────┘

┌──────┬────────┬──────┐
│ id   │ name   │ age  │
│ long │ string │ long │
├──────┼────────┼──────┤
│    1 │ alice  │   10 │
├──────┴────────┴──────┤
│ 1 rows               │
└──────────────────────┘

[ok]:    1 was equal to 1 (01JAXGC09FFAZ0ZNHAWVWZSBJ4.wv:8:13)
[ok]:    [[1, alice, 10]] was equal to [[1, alice, 10]] (01JAXGC09FFAZ0ZNHAWVWZSBJ4.wv:9:13)
```

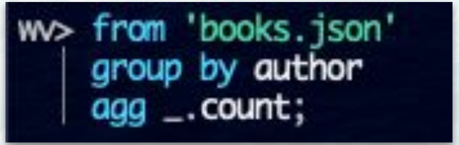
Regular Query Result

Debug Query Result

Test Result

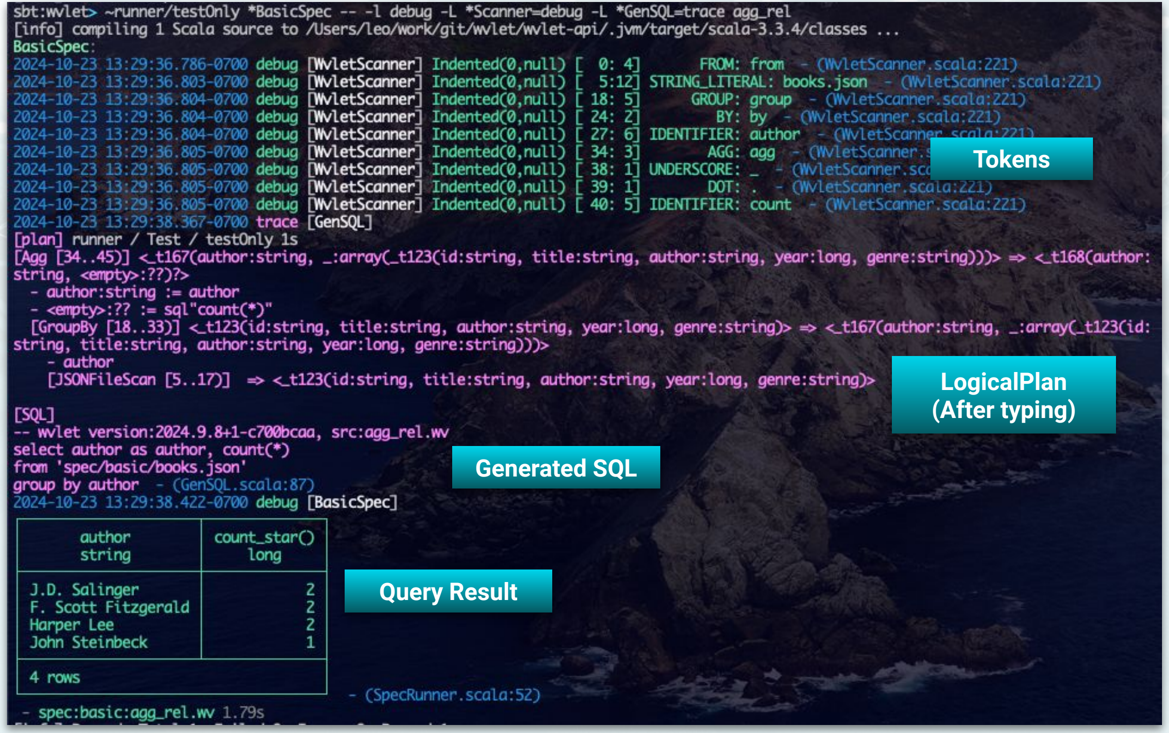
Wvlet Internals: Compiler

- Compiler phases: Parser -> SymbolLabeler -> TypeResolver -> Rewriter -> ExecutionPlanner -> GenSQL
  - CompilationUnit holds the source text, untyped logical plan, typed logical plan, execution plan, etc.
- With AirSpec testing library, you can adjust the log levels of individual components
  - -L (class name pattern)=(log level) option



```
wv> from 'books.json'
  | group by author
  | agg _.count;
```

Query (.wv)



```
sbt:wvlet> ~runner/testOnly *BasicSpec -- -l debug -L *Scanner=debug -L *GenSQL=trace agg_rel
[info] compiling 1 Scala source to /Users/leo/work/git/wvlet/wvlet-api/.jvm/target/scala-3.3.4/classes ...
BasicSpec:
2024-10-23 13:29:36.786-0700 debug [WvletScanner] Indented(0,null) [  0: 4]       FROM: from  - (WvletScanner.scala:221)
2024-10-23 13:29:36.803-0700 debug [WvletScanner] Indented(0,null) [  5:12] STRING_LITERAL: books.json  - (WvletScanner.scala:221)
2024-10-23 13:29:36.804-0700 debug [WvletScanner] Indented(0,null) [ 18: 5]      GROUP: group  - (WvletScanner.scala:221)
2024-10-23 13:29:36.804-0700 debug [WvletScanner] Indented(0,null) [ 24: 2]         BY: by  - (WvletScanner.scala:221)
2024-10-23 13:29:36.804-0700 debug [WvletScanner] Indented(0,null) [ 27: 6] IDENTIFIER: author  - (WvletScanner.scala:221)
2024-10-23 13:29:36.805-0700 debug [WvletScanner] Indented(0,null) [ 34: 3]        AGG: agg  - (WvletScanner.s
2024-10-23 13:29:36.805-0700 debug [WvletScanner] Indented(0,null) [ 38: 1] UNDERSCORE: _  - (WvletScanner.sc
2024-10-23 13:29:36.805-0700 debug [WvletScanner] Indented(0,null) [ 39: 1]        DOT: .  - (WvletScanner.scala:221)
2024-10-23 13:29:36.805-0700 debug [WvletScanner] Indented(0,null) [ 40: 5] IDENTIFIER: count  - (WvletScanner.scala:221)
2024-10-23 13:29:38.367-0700 trace [GenSQL]
[plan] runner / Test / testOnly 1s
[Agg [34..45)] <_t167(author:string, _:array(_t123(id:string, title:string, author:string, year:long, genre:string)))> => <_t168(author:
string, <empty>:??)?>
  - author:string := author
  - <empty>:?? := sql"count(*)"
  [GroupBy [18..33)] <_t123(id:string, title:string, author:string, year:long, genre:string)> => <_t167(author:string, _:array(_t123(id:
string, title:string, author:string, year:long, genre:string)))>
    - author
    [JSONFileScan [5..17)]  => <_t123(id:string, title:string, author:string, year:long, genre:string)>

[SQL]
-- wvlet version:2024.9.8+1-c700bcaa, src:agg_rel.wv
select author as author, count(*)
from 'spec/basic/books.json'
group by author  - (GenSQL.scala:87)
2024-10-23 13:29:38.422-0700 debug [BasicSpec]
┌─────────────────────┬───────────────┐
│       author        │ count_star()  │
│       string        │     long      │
├─────────────────────┼───────────────┤
│ J.D. Salinger       │             2 │
│ F. Scott Fitzgerald │             2 │
│ Harper Lee          │             2 │
│ John Steinbeck      │             1 │
├─────────────────────┴───────────────┤
│ 4 rows                              │
└─────────────────────────────────────┘
                                        - (SpecRunner.scala:52)
 - spec:basic:agg_rel.wv 1.79s
```

Tokens

LogicalPlan (After typing)

Generated SQL

Query Result

Wvlet Development Roadmap

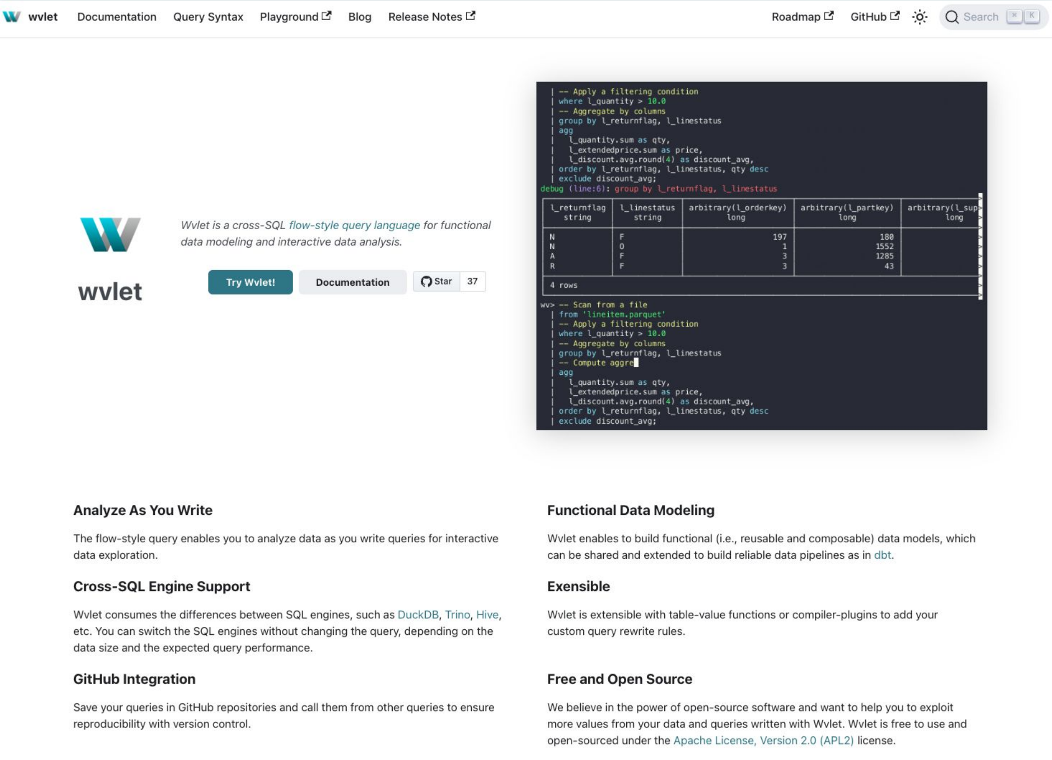
- Roadmap is maintained at GitHub Project page.
  - Planning milestones for about every 3 months
  - Versions will be YYYY.(milestone month).(patch) (See #170 for the versioning scheme)
- Release 2024.9
  - ✅ Flow-style query language design and compiler
    - Including scanner, parser, typer, tree rewrite framework, execution planner, SQL generator, etc.
  - ✅ DBMS Connector (DuckDB, Trino)
  - ✅ wv: Interactive command-line editor (REPL)
  - ✅ Installer (Homebrew)
  - ✅ Web UI, Playground (Monaco Editor with DuckDB-Wasm)
- Release 2025.1
  - Model management
    - Generate dependent model materialization plan, like dbt
    - GitHub integration
  - Compiler plugins for advance optimization
    - Incremental processing, query fusion, etc.
  - Language SDKs
    - Python, Rust, C/C++, Java, etc.
  - SQL to Wvlet converter

Summary

- Designed Wvlet, a new open-source flow-style query language

- Addresses challenges in 50-year-old design of SQL
  - Leverage good sides of SQL
    - natural-language like syntax
  - Intuitive syntax for data flow
  - Reusability
  - Extensibility

- Website: https://wvlet.org/
- GitHub: https://github.com/wvlet/wvlet



Wvlet is a cross-SQL flow-style query language for functional data modeling and interactive data analysis.

```
  | -- Apply a filtering condition
  | where l_quantity > 10.0
  | -- Aggregate by columns
  | group by l_returnflag, l_linestatus
  | agg
  |   l_quantity.sum as qty,
  |   l_extendedprice.sum as price,
  |   l_discount.avg.round(4) as discount_avg,
  | order by l_returnflag, l_linestatus, qty desc
  | exclude discount_avg;
debug (line:6): group by l_returnflag, l_linestatus
┌─────────────┬─────────────┬────────────────────┬───────────────────┬───────────────
│ l_returnflag│ l_linestatus│ arbitrary(l_orderkey)│ arbitrary(l_partkey)│ arbitrary(l_sup
│   string    │   string    │        long         │       long        │       long
├─────────────┼─────────────┼────────────────────┼───────────────────┼───────────────
│ N           │ F           │                 197│                180│
│ N           │ O           │                   1│               1552│
│ A           │ F           │                   3│               1285│
│ R           │ F           │                   3│                 43│
├─────────────┴─────────────┴────────────────────┴───────────────────┴───────────────
│ 4 rows
└──────────────────────────────────────────────────────────────────────────────────
wv> -- Scan from a file
  | from 'lineitem.parquet'
  | -- Apply a filtering condition
  | where l_quantity > 10.0
  | -- Aggregate by columns
  | group by l_returnflag, l_linestatus
  | -- Compute aggre
  | agg
  |   l_quantity.sum as qty,
  |   l_extendedprice.sum as price,
  |   l_discount.avg.round(4) as discount_avg,
  | order by l_returnflag, l_linestatus, qty desc
  | exclude discount_avg;
```

Analyze As You Write

The flow-style query enables you to analyze data as you write queries for interactive data exploration.

Cross-SQL Engine Support

Wvlet consumes the differences between SQL engines, such as DuckDB, Trino, Hive, etc. You can switch the SQL engines without changing the query, depending on the data size and the expected query performance.

GitHub Integration

Save your queries in GitHub repositories and call them from other queries to ensure reproducibility with version control.

Functional Data Modeling

Wvlet enables to build functional (i.e., reusable and composable) data models, which can be shared and extended to build reliable data pipelines as in dbt.

Exensible

Wvlet is extensible with table-value functions or compiler-plugins to add your custom query rewrite rules.

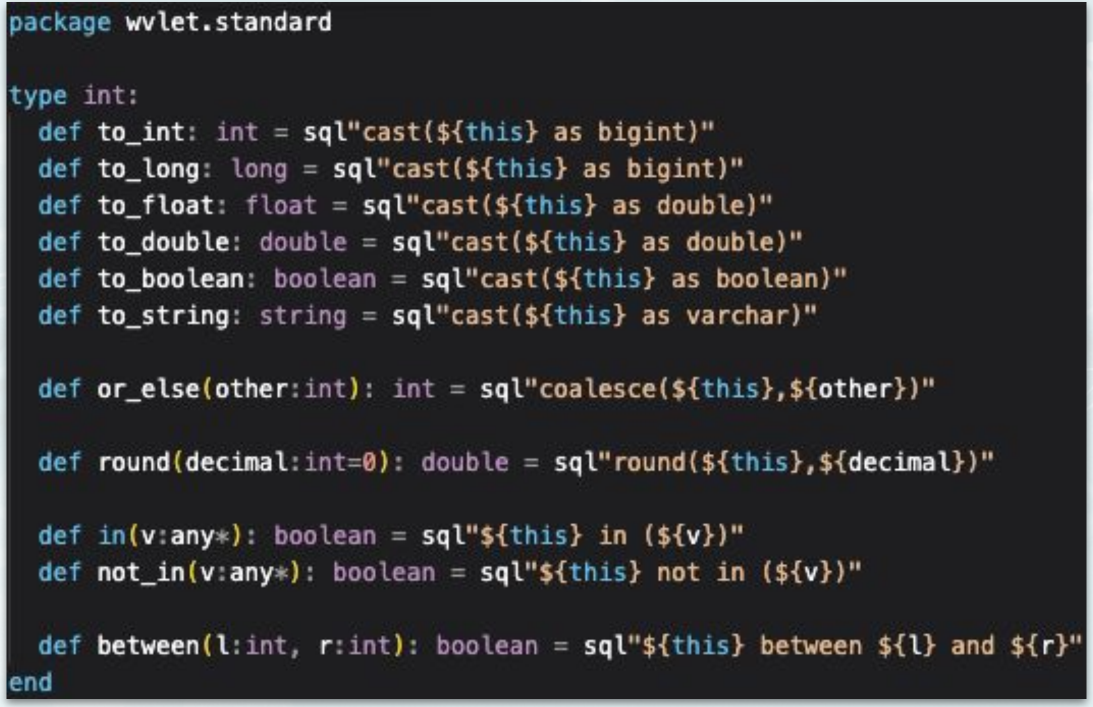
Free and Open Source

We believe in the power of open-source software and want to help you to exploit more values from your data and queries written with Wvlet. Wvlet is free to use and open-sourced under the Apache License, Version 2.0 (APL2) license.

Appendix

Wvlet Internals: Standard Library

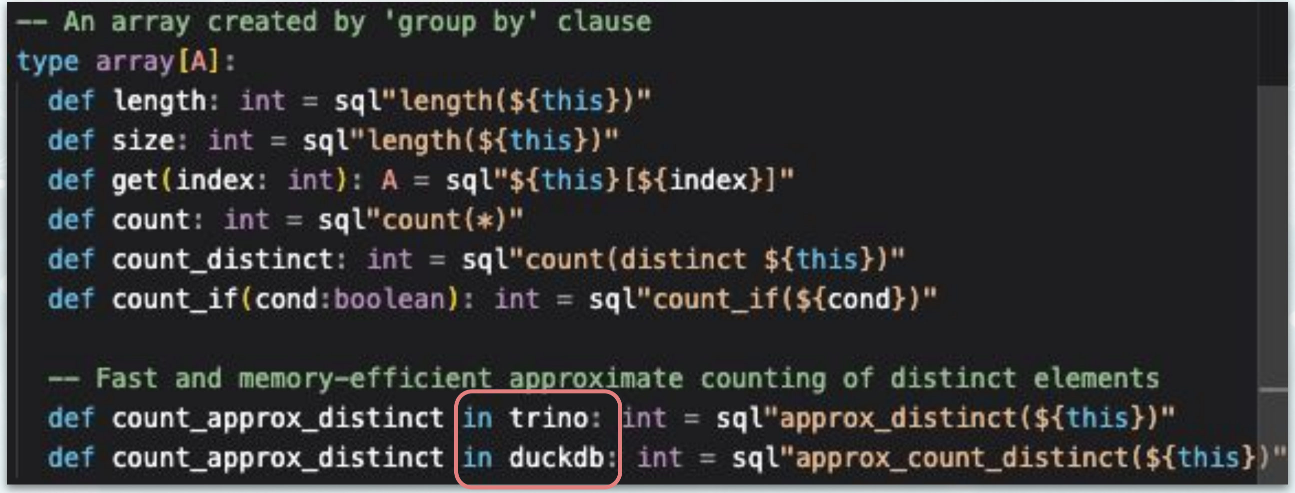
- Wvlet defines standard data types and functions, which define how to generate SQL
  - example: x.to_int.round(1) => cast(x as bigint).round(1) => round(cast(x as bigint),1)



```
package wvlet.standard

type int:
  def to_int: int = sql"cast(${this} as bigint)"
  def to_long: long = sql"cast(${this} as bigint)"
  def to_float: float = sql"cast(${this} as double)"
  def to_double: double = sql"cast(${this} as double)"
  def to_boolean: boolean = sql"cast(${this} as boolean)"
  def to_string: string = sql"cast(${this} as varchar)"

  def or_else(other:int): int = sql"coalesce(${this},${other})"

  def round(decimal:int=0): double = sql"round(${this},${decimal})"

  def in(v:any*): boolean = sql"${this} in (${v})"
  def not_in(v:any*): boolean = sql"${this} not in (${v})"

  def between(l:int, r:int): boolean = sql"${this} between ${l} and ${r}"
end
```

Wvlet Internals: DBMS Specific Functions

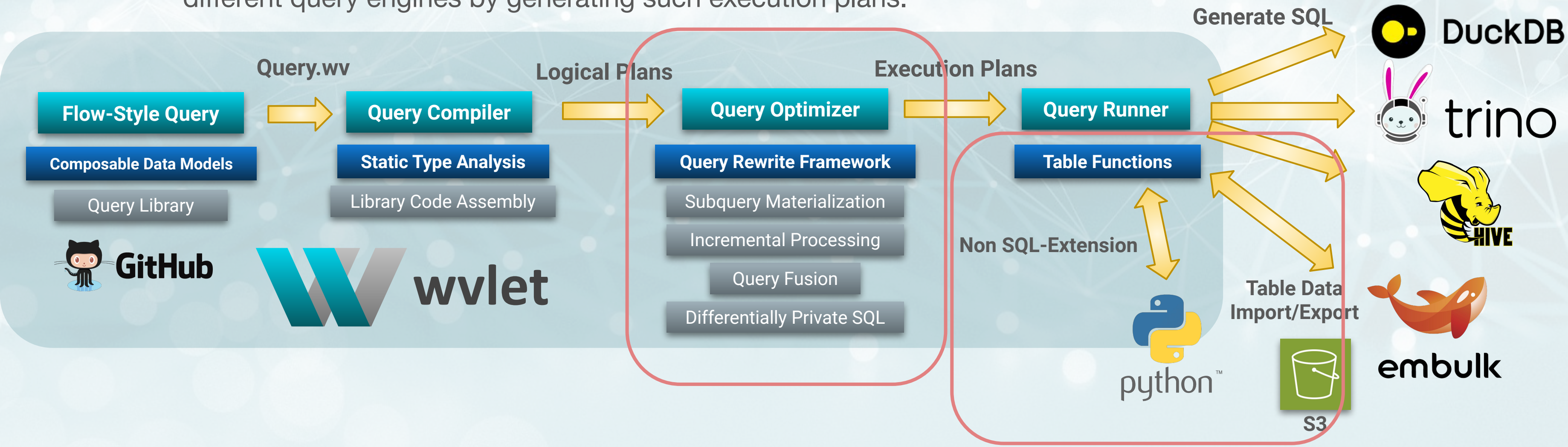
- Function can have different implementations depending on the target databases (Trino/Hive/DuckDB, etc.)
  - x.count_approx_distinct
    - Trino: approx_distinct(x)
    - DuckDB: approx_count_distinct(x)
- Works for consuming the differences between SQL dialects and UDFs



```
-- An array created by 'group by' clause
type array[A]:
  def length: int = sql"length(${this})"
  def size: int = sql"length(${this})"
  def get(index: int): A = sql"${this}[${index}]"
  def count: int = sql"count(*)"
  def count_distinct: int = sql"count(distinct ${this})"
  def count_if(cond:boolean): int = sql"count_if(${cond})"

  -- Fast and memory-efficient approximate counting of distinct elements
  def count_approx_distinct in trino: int = sql"approx_distinct(${this})"
  def count_approx_distinct in duckdb: int = sql"approx_count_distinct(${this})"
```

Extending Wvlet

- **Compiler plugins** (to be designed #185)
  - Add a custom rule set to optimize logical plans and execution plans.
    - Optimization rules: Query fusion optimization, subquery materialization, incremental processing, etc.
  - Security rules (e.g., forbidding local file access operators at the cloud environment)
- **Table functions** receive table-value data and output table value data
  - Not limited to SQL, we will be able to invoke ML algorithms, calling Embulk, or issuing SQL queries to different query engines by generating such execution plans.



Query.wv
Logical Plans
Execution Plans
Generate SQL

Flow-Style Query
Composable Data Models
Query Library

Query Compiler
Static Type Analysis
Library Code Assembly

Query Optimizer
Query Rewrite Framework
Subquery Materialization
Incremental Processing
Query Fusion
Differentially Private SQL

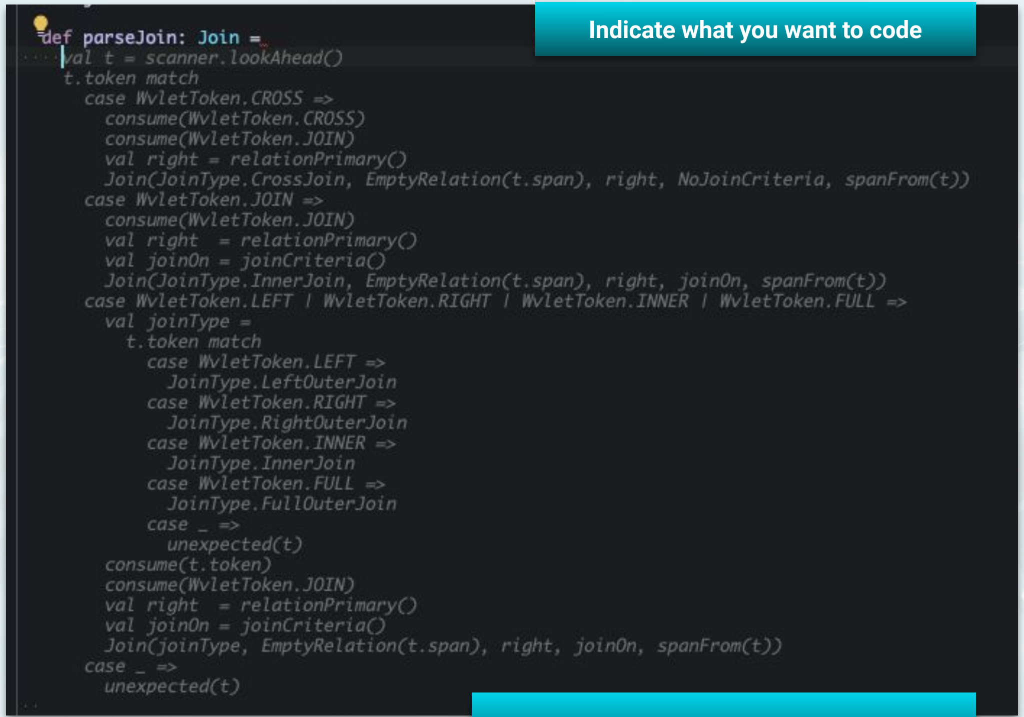
Query Runner
Table Functions
Non SQL-Extension
Table Data Import/Export

Writing A Compiler From Scratch? Leverage LLMs

- Developing a compiler is a challenge with known difficulties and known solutions
  - = solved problem

- By leveraging LLM-based tools like GitHub Copilot, we can accelerate the development

- LLMs excel at utilizing well-known solutions from open-source code.



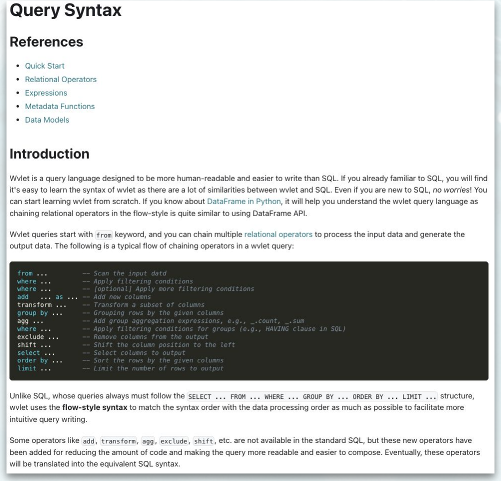
Indicate what you want to code

```
def parseJoin: Join =
  val t = scanner.lookAhead()
  t.token match
    case WvletToken.CROSS =>
      consume(WvletToken.CROSS)
      consume(WvletToken.JOIN)
      val right = relationPrimary()
      Join(JoinType.CrossJoin, EmptyRelation(t.span), right, NoJoinCriteria, spanFrom(t))
    case WvletToken.JOIN =>
      consume(WvletToken.JOIN)
      val right  = relationPrimary()
      val joinOn = joinCriteria()
      Join(JoinType.InnerJoin, EmptyRelation(t.span), right, joinOn, spanFrom(t))
    case WvletToken.LEFT | WvletToken.RIGHT | WvletToken.INNER | WvletToken.FULL =>
      val joinType =
        t.token match
          case WvletToken.LEFT =>
            JoinType.LeftOuterJoin
          case WvletToken.RIGHT =>
            JoinType.RightOuterJoin
          case WvletToken.INNER =>
            JoinType.InnerJoin
          case WvletToken.FULL =>
            JoinType.FullOuterJoin
          case _ =>
            unexpected(t)
      consume(t.token)
      consume(WvletToken.JOIN)
      val right  = relationPrimary()
      val joinOn = joinCriteria()
      Join(joinType, EmptyRelation(t.span), right, joinOn, spanFrom(t))
    case _ =>
      unexpected(t)
```

GitHub Copilot suggests code in a second

Wvlet is easy to learn, even for Large Language Models (LLMs)

- An LLM Agent, created from Wvlet documentation, can successfully convert TPC-H SQL queries into Wvlet syntax
  - Wvlet: Query Syntax documentation (link)



Query Syntax

References

- Quick Start
- Relational Operators
- Expressions
- Metadata Functions
- Data Models

Introduction

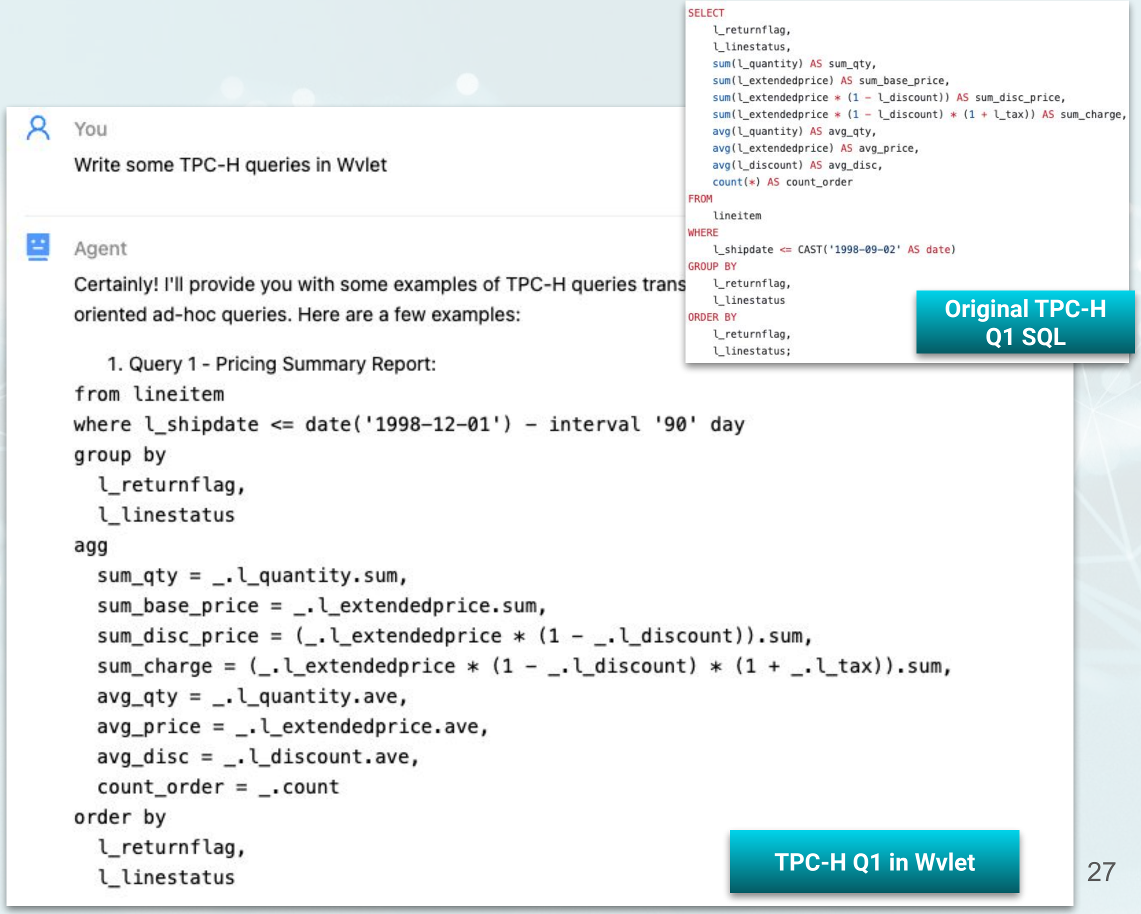
Wvlet is a query language designed to be more human-readable and easier to write than SQL. If you already familiar to SQL, you will find it's easy to learn the syntax of wvlet as there are a lot of similarities between wvlet and SQL. Even if you are new to SQL, *no worries*! You can start learning wvlet from scratch. If you know about DataFrame in Python, it will help you understand the wvlet query language as chaining relational operators in the flow-style is quite similar to using DataFrame API.

Wvlet queries start with `from` keyword, and you can chain multiple relational operators to process the input data and generate the output data. The following is a typical flow of chaining operators in a wvlet query:

```
from ...          -- Scan the input datd
where ...         -- Apply filtering conditions
where ...         -- [optional] Apply more filtering conditions
add   ... as ... -- Add new columns
transform ...     -- Transform a subset of columns
group by ...      -- Grouping rows by the given columns
agg ...           -- Add group aggregation expressions, e.g., _.count, _.sum
where ...         -- Apply filtering conditions for groups (e.g., HAVING clause in SQL)
exclude ...       -- Remove columns from the output
shift ...         -- Shift the column position to the left
select ...        -- Select columns to output
order by ...      -- Sort the rows by the given columns
limit ...         -- Limit the number of rows to output
```

Unlike SQL, whose queries always must follow the `SELECT ... FROM ... WHERE ... GROUP BY ... ORDER BY ... LIMIT ...` structure, wvlet uses the **flow-style syntax** to match the syntax order with the data processing order as much as possible to facilitate more intuitive query writing.

Some operators like `add`, `transform`, `agg`, `exclude`, `shift`, etc. are not available in the standard SQL, but these new operators have been added for reducing the amount of code and making the query more readable and easier to compose. Eventually, these operators will be translated into the equivalent SQL syntax.



You

Write some TPC-H queries in Wvlet

Agent

Certainly! I'll provide you with some examples of TPC-H queries trans

oriented ad-hoc queries. Here are a few examples:

1. Query 1 - Pricing Summary Report:

```
from lineitem
where l_shipdate <= date('1998-12-01') - interval '90' day
group by
  l_returnflag,
  l_linestatus
agg
  sum_qty = _.l_quantity.sum,
  sum_base_price = _.l_extendedprice.sum,
  sum_disc_price = (_.l_extendedprice * (1 - _.l_discount)).sum,
  sum_charge = (_.l_extendedprice * (1 - _.l_discount) * (1 + _.l_tax)).sum,
  avg_qty = _.l_quantity.ave,
  avg_price = _.l_extendedprice.ave,
  avg_disc = _.l_discount.ave,
  count_order = _.count
order by
  l_returnflag,
  l_linestatus
```

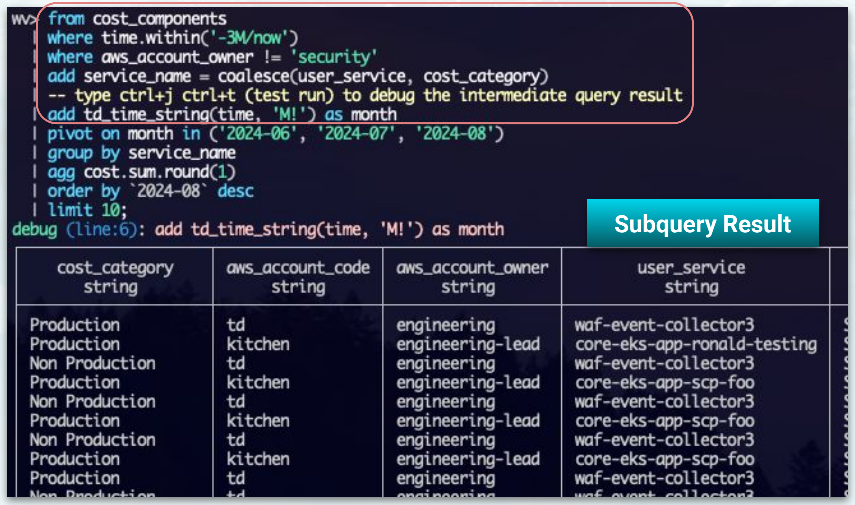
TPC-H Q1 in Wvlet

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) AS sum_qty,
    sum(l_extendedprice) AS sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    avg(l_quantity) AS avg_qty,
    avg(l_extendedprice) AS avg_price,
    avg(l_discount) AS avg_disc,
    count(*) AS count_order
FROM
    lineitem
WHERE
    l_shipdate <= CAST('1998-09-02' AS date)
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,
    l_linestatus;
```

Original TPC-H Q1 SQL

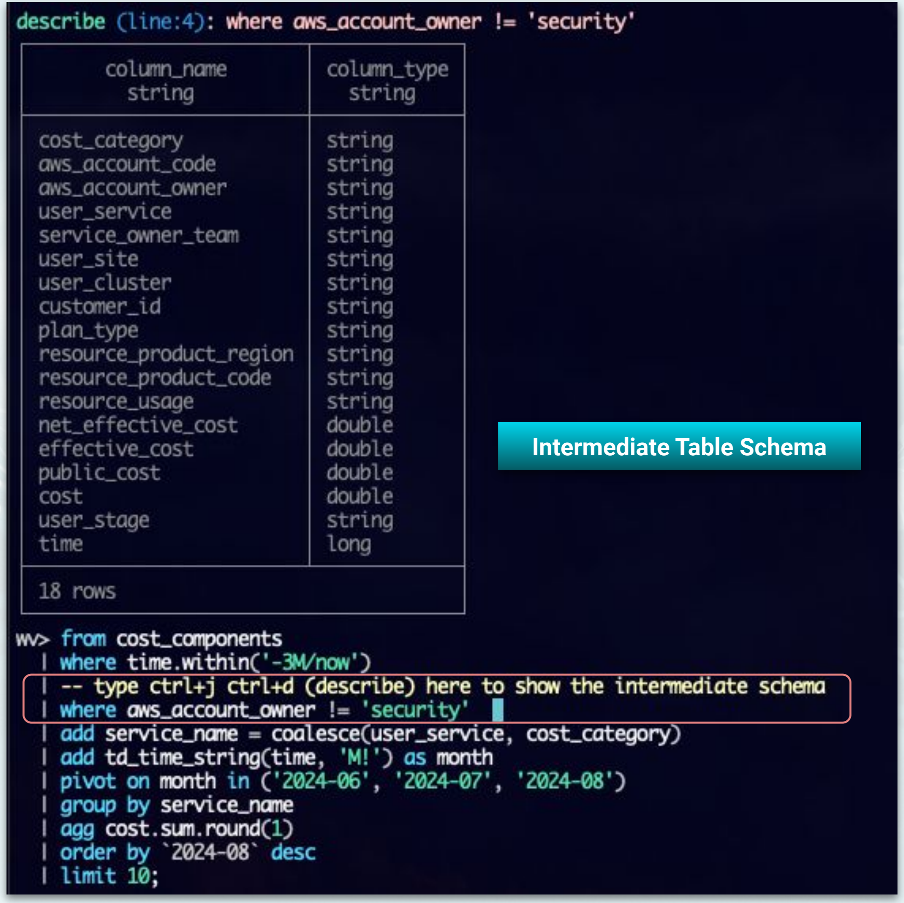
wv: Interactive Query Editor

- Available with brew install wvlet/wvlet/wvlet
    - Supports Trino, DuckDB via profile settings
- Shortcut keys for checking intermediate schema and results
    - ctrl-j, ctrl-d (describe the schema at the line)
    - ctrl-j, ctrl-t (test run the subquery upto the line)
    - ctrl-j, ctrl-r (run the whole query)



```
wv> from cost_components
  | where time.within('-3M/now')
  | where aws_account_owner != 'security'
  | add service_name = coalesce(user_service, cost_category)
  | -- type ctrl+j ctrl+t (test run) to debug the intermediate query result
  | add td_time_string(time, 'M!') as month
  | pivot on month in ('2024-06', '2024-07', '2024-08')
  | group by service_name
  | agg cost.sum.round(1)
  | order by `2024-08` desc
  | limit 10;
debug (line:6): add td_time_string(time, 'M!') as month
┌────────────────┬──────────────────┬───────────────────┬──────────────────────────────┐
│ cost_category  │ aws_account_code │ aws_account_owner │         user_service         │
│     string     │      string      │      string       │            string            │
├────────────────┼──────────────────┼───────────────────┼──────────────────────────────┤
│ Production     │ td               │ engineering       │ waf-event-collector3         │
│ Production     │ kitchen          │ engineering-lead  │ core-eks-app-ronald-testing  │
│ Non Production │ td               │ engineering       │ waf-event-collector3         │
│ Production     │ kitchen          │ engineering-lead  │ core-eks-app-scp-foo         │
│ Non Production │ td               │ engineering       │ waf-event-collector3         │
│ Production     │ kitchen          │ engineering-lead  │ core-eks-app-scp-foo         │
│ Non Production │ td               │ engineering       │ waf-event-collector3         │
│ Production     │ kitchen          │ engineering-lead  │ core-eks-app-scp-foo         │
│ Production     │ td               │ engineering       │ waf-event-collector3         │
```

Subquery Result



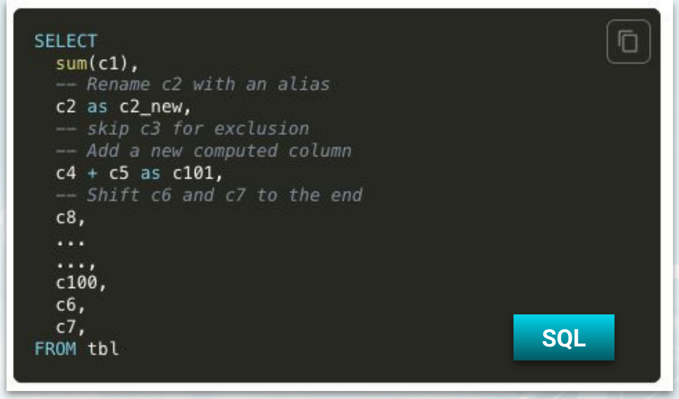
```
describe (line:4): where aws_account_owner != 'security'
┌─────────────────────────┬─────────────┐
│       column_name       │ column_type │
│         string          │   string    │
├─────────────────────────┼─────────────┤
│ cost_category           │ string      │
│ aws_account_code        │ string      │
│ aws_account_owner       │ string      │
│ user_service            │ string      │
│ service_owner_team      │ string      │
│ user_site               │ string      │
│ user_cluster            │ string      │
│ customer_id             │ string      │
│ plan_type               │ string      │
│ resource_product_region │ string      │
│ resource_product_code   │ string      │
│ resource_usage          │ string      │
│ net_effective_cost      │ double      │
│ effective_cost          │ double      │
│ public_cost             │ double      │
│ cost                    │ double      │
│ user_stage              │ string      │
│ time                    │ long        │
├─────────────────────────┴─────────────┤
│ 18 rows                               │
└───────────────────────────────────────┘
wv> from cost_components
  | where time.within('-3M/now')
  | -- type ctrl+j ctrl+d (describe) here to show the intermediate schema
  | where aws_account_owner != 'security'
  | add service_name = coalesce(user_service, cost_category)
  | add td_time_string(time, 'M!') as month
  | pivot on month in ('2024-06', '2024-07', '2024-08')
  | group by service_name
  | agg cost.sum.round(1)
  | order by `2024-08` desc
  | limit 10;
```

Intermediate Table Schema

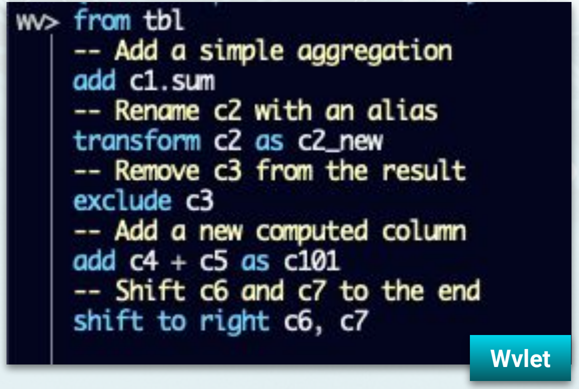
Design Philosophy of Wvlet

- Link: https://wvlet.org/wvlet/docs/internal/design

- Matching syntax order with data flow
  - Typing from left-to-right, top-to-bottom order
- Use only lowercase keywords
  - Forbid mixing SELECT and select
- One operation at a time
  - SQL's SELECT operator is too powerful
  - Breaking down SELECT statement into simple operators (add, transform, exclude, shift, etc)
- Human and machine friendly-syntax
  - Query lines form subqueries
    - Trailing comma support is essential to correctly extract subqueries in an executable form
  - Enable in-query data/schema inspection



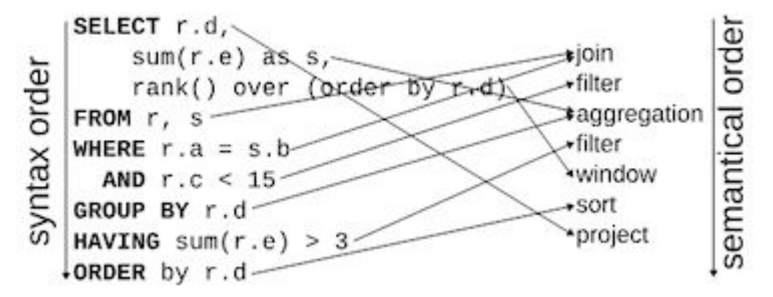
```
SELECT
  sum(c1),
  -- Rename c2 with an alias
  c2 as c2_new,
  -- skip c3 for exclusion
  -- Add a new computed column
  c4 + c5 as c101,
  -- Shift c6 and c7 to the end
  c8,
  ...
  ...,
  c100,
  c6,
  c7,
FROM tbl
```



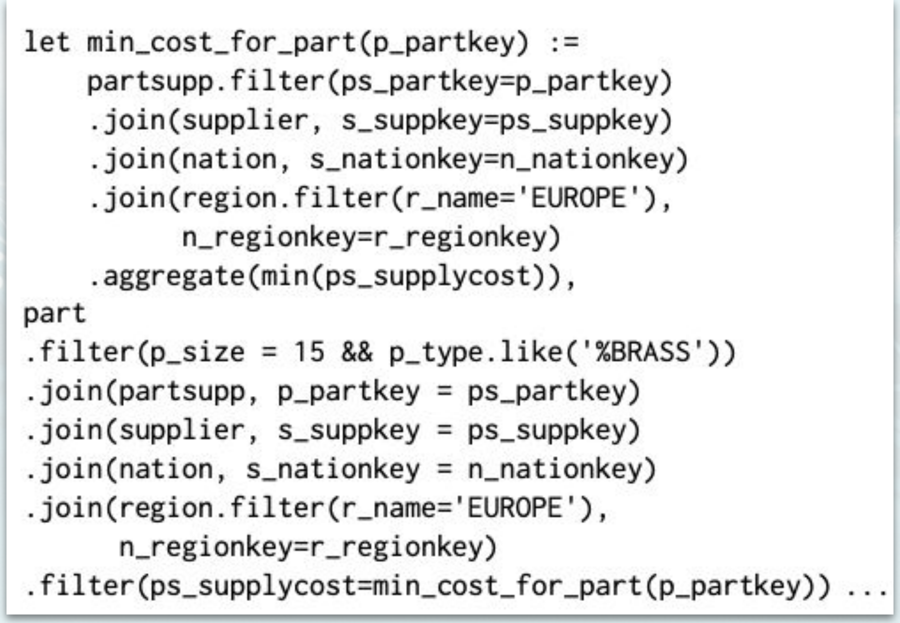
```
wv> from tbl
    -- Add a simple aggregation
    add c1.sum
    -- Rename c2 with an alias
    transform c2 as c2_new
    -- Remove c3 from the result
    exclude c3
    -- Add a new computed column
    add c4 + c5 as c101
    -- Shift c6 and c7 to the end
    shift to right c6, c7
```

Related Work: SaneQL (Thomas Neumann, et al. 2024)

- The syntactic order of SQL doesn't match with the semantic order of data processing
- Redesigned SQL by using a function chain syntax
  - A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language (CIDR 24)



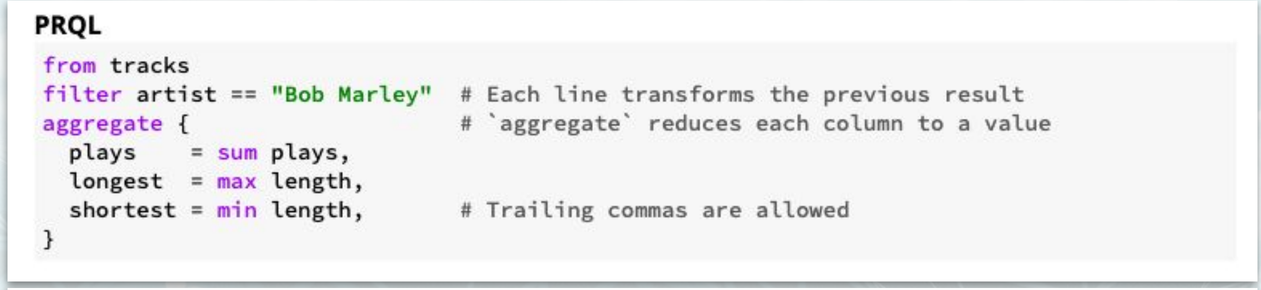
```
SELECT r.d,
    sum(r.e) as s,
    rank() over (order by r.d)
FROM r, s
WHERE r.a = s.b
  AND r.c < 15
GROUP BY r.d
HAVING sum(r.e) > 3
ORDER by r.d
```



```
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)) ...
```

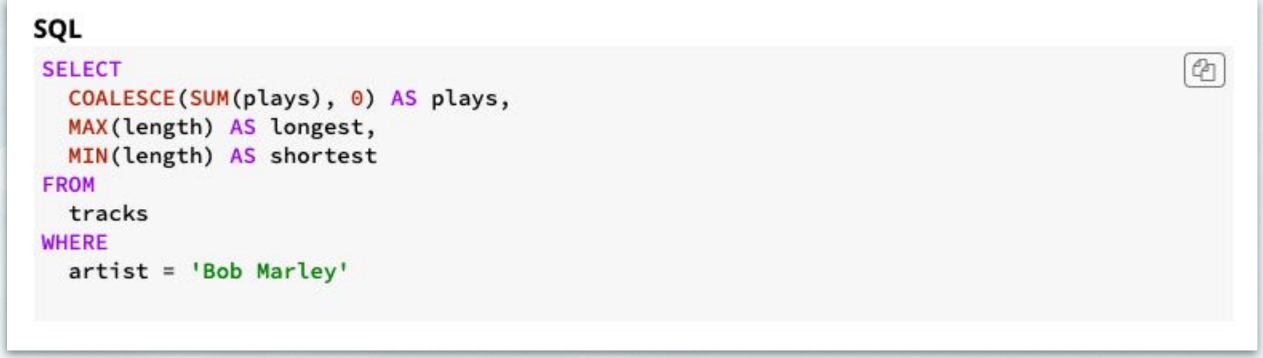
Related Work: PRQL (Prequel)

- Pipelined Relational Query Language (Since 2022) https://prql-lang.org/
- Written in Rust, compiling PRQL to SQL
  - Syntax is a bit far from SQL, more like DataFrame API or programming language syntax



PRQL

```
from tracks
filter artist == "Bob Marley"  # Each line transforms the previous result
aggregate {                    # `aggregate` reduces each column to a value
  plays    = sum plays,
  longest  = max length,
  shortest = min length,       # Trailing commas are allowed
}
```



SQL

```
SELECT
  COALESCE(SUM(plays), 0) AS plays,
  MAX(length) AS longest,
  MIN(length) AS shortest
FROM
  tracks
WHERE
  artist = 'Bob Marley'
```

TREASURE DATA