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

Thomas Neumann
Viktor Leis

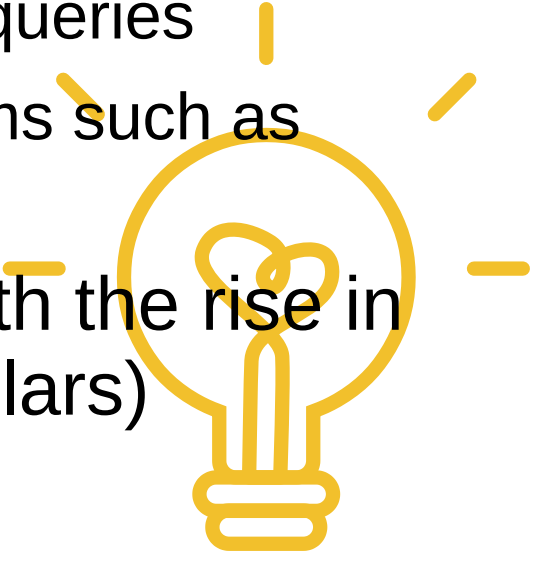
Nikolas Vattis
Antonis Katsiantonis

CONTENT

- Introduction
- A critique of Modern SQL
- SaneQL
- Conclusion

INTRODUCTION

- As of today, SQL is the predominant query language
- However, SQL faces design problems
  - Problem 1 – English syntax, easy to read for simple queries, but hard to read for more complex queries
  - Problem 2 – Lacks programming mechanisms such as Abstraction, Modularity, Extensibility
- The dominance of SQL may be at risk with the rise in popularity of dataframe APIs (pandas, polars)



CONTRIBUTIONS

- Provide a detailed critique on the modern SQL language
- Introduce **SaneQL** (Simple ANd Expressive Query Language)

A CRITIQUE OF MODERN SQL

- Modern SQL is critiqued using query examples
- Authors have collected 130998 queries from over 1000 students
- 38% resulted in an error on first execution
- Vast majority of failed executions were at compile-time

UNHELPFUL ERROR MESSAGES

- Most common error messages are not very helpful
- Error messages do not specify the problem, or how to fix it

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

SYNTACTICALLY DIFFICULT CONSTRUCTS

- WITH
  - Only 45% of queries containing WITH are successful
- VALUES
  - Only 40% of queries containing VALUES are successful

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

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

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

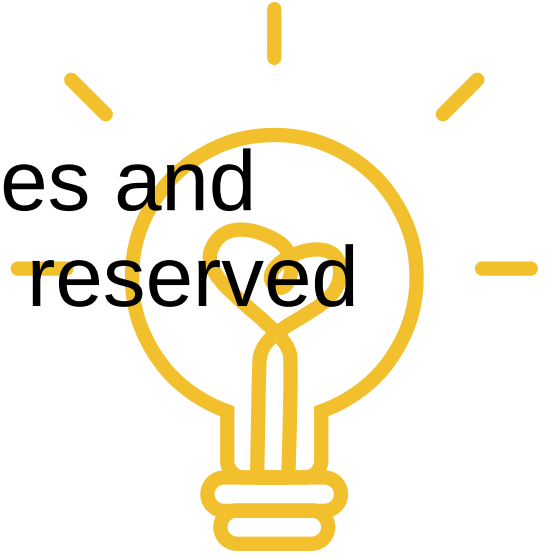
```
SELECT * FROM (VALUES ('x',5), ('y',2)) AS r(a,b)
```

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

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

TOO MANY KEYWORDS

- The SQL parser heavily relies on reserved keywords
- 401 English words reserved, representing 18% of all English word usage
- This causes invalid accesses to tables and attributes which are named after the reserved keywords



IMPLICIT JOINS

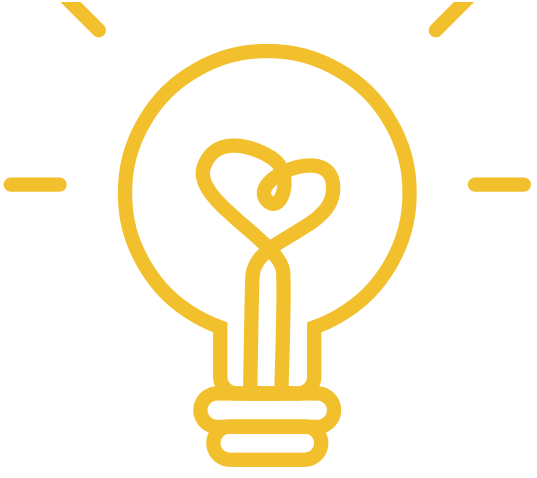
- Can accidentally create huge, slow queries if a join condition is forgotten.
- SQL allows joins using just a comma.
- This can overload systems and even give incorrect results.
- **Making joins so easy to write is a risky design choice.**

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

EXPLICIT JOINS

- Syntax is crucial for outer joins since their results depend on the join order.
- Two similar-looking queries can produce different results based on where the ON clause is placed.
- This makes understanding SQL join behavior tricky, as the syntax doesn't clearly show the execution order.

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



“GROUP” OPERATOR

- Another common operation is grouping.
- Grouped attributes appear twice.
- 9% of all queries failing with a “column must appear in the GROUP BY clause” error

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

“HAVING” OPERATOR

- A common SQL mistake (1.9% of errors) is filtering aggregates in the WHERE clause instead of using HAVING.
- Subqueries could work, SQL provides HAVING specifically for avoiding subqueries

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

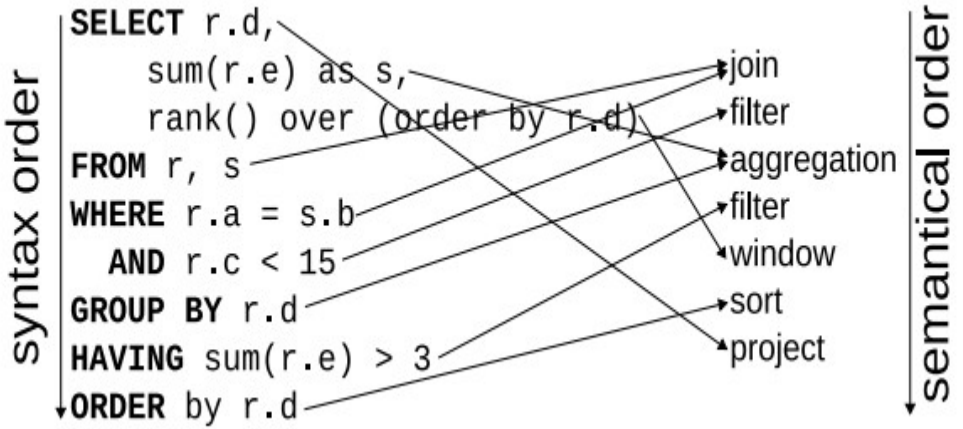
“OVER” OPERATOR

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

- Another keyword which can be confusing
- OVER operator is executed later based on the semantic order
- This means that the WHERE operator does not have access to the OVER operator
- User must use a workaround by using subqueries

SYNTACTIC / SEMANTIC ORDERING

- Syntactic order doesn't match its actual execution order, making queries harder to understand.
- The **SELECT** clause is a clear example, as its parts execute at different stages.



LACK OF PORTABILITY

- SQL is an official ISO standard.
- However…
  - Very hard to write portable SQL queries.
  - Systems implement non-portable expression libraries and language extensions, which further fragments the language.
  - Systems choose to deviate from the standard.

LESSONS

- **Irregular Syntax Causes Big Problems:**
  - English-inspired syntax leads to a complex and arbitrary grammar, making the language difficult to learn.
  - Languages have to be designed with extensibility and abstraction mechanisms in mind.
- **Semantic Operator Ordering Should Be Explicit:**
  - To compose accurate SQL queries, it is necessary to comprehend the implicit ordering semantics of each construct.
  - It would be better for query languages to make the semantical and syntactical order identical

SaneQL: TOWARDS SIMPLE AND EXPRESSIVE QUERIES

- New query language, called the Simple ANd Expressive Query Language (SaneQL).
  - Nicer and more systematic way to expressive queries.
  - Modularity allows reusing logic across queries.

FOUNDATION – RELATIONAL ALGEBRA

- SaneQL is based on relational algebra, allowing users to construct queries using relational operators.
- **Query Optimizer:**
  - Just like in SQL, SaneQL's algebraic expressions are optimized into more efficient execution plans.
- **New Data Categories:**
  - Introduces expressions, symbols and lists of these elements.

QUERY EXPRESSION – PIPELINING

- SaneQL supports Uniform Function Call Syntax (UFCS).
  - We can use a dot to pass a value as first argument of the next call.
  - The dot notation is usually preferable as it preserves locality in the query text.

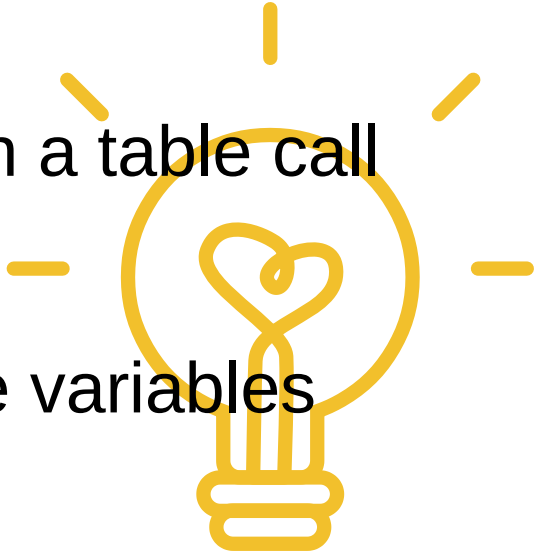
SYNTAX – CALLS, KEYWORDS, LISTS

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

- Operations are performed by invoking functions
- Named parameters inside functions help to keep invocations readable
- Curly braces are used to denote lists. Used for grouping and aggregation

LANGUAGE FEATURES

- Scoping
  - If a tuple is referenced multiple times, it can be given an alias using the as(…) operator
- Inline Tables
  - Constant tables that are constructed with a table call
- Let construct
  - Intermediate results can be stored inside variables



MODULARITY AND EXTENSIBILITY

- Scalar arguments
  - SaneQL provides the additional feature of parameterizing queries for modular query execution
- Expression arguments
  - Modularize queries with the ability to pass expressions as arguments

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

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

RELATED WORK

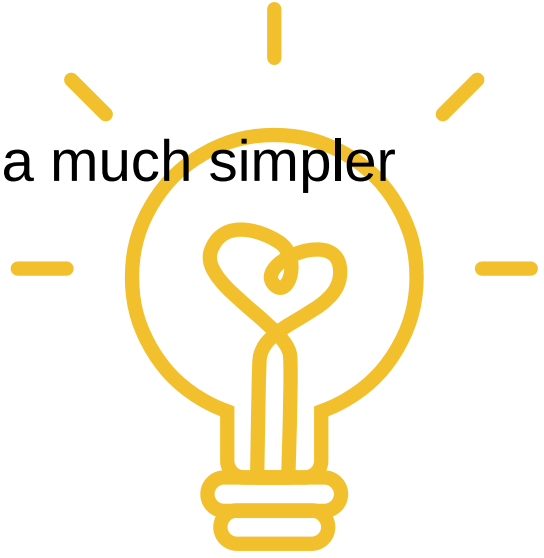
- How We Got Here
  - Over five decades, incremental additions like subqueries, outer joins, and window functions transformed SQL into a powerful but increasingly complex language.
- SQL Critiques
  - SQL has been receiving critique since it’s release
- New query languages
  - There have been many attempts for a better language
  - SaneQL distinguishes itself from other attempts due to it’s abstraction capabilities

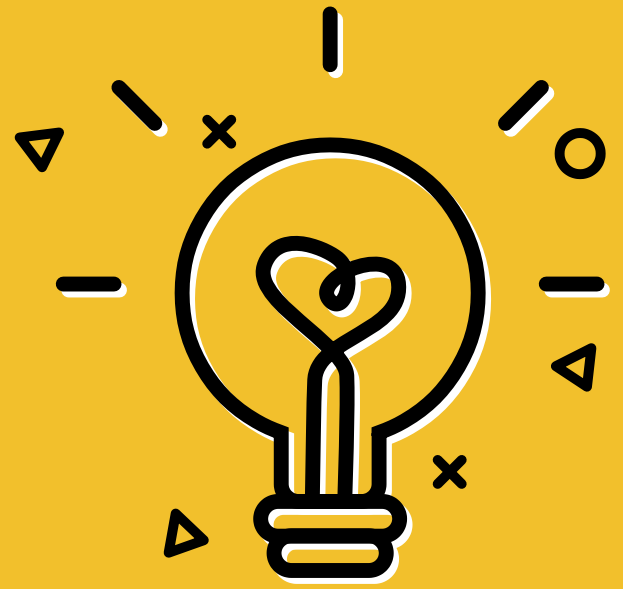
FUTURE WORK

- Authors’ future work includes exploring SaneQL’s embedding into host languages, integrating operator hints for query optimization, developing interactive query construction interfaces, and introducing a general macro system for advanced abstractions.

CONCLUSIONS

- Despite being the dominant language, SQL has major weaknesses
  - Hard to learn and difficult to debug for beginners
  - Lack of abstraction features for advanced users
- SaneQL
  - Replaces the irregular pseudo-English syntax with a much simpler regular syntax
  - Explicitly orders the relational operations
  - Preserves SQL’s core ideas





THANK YOU