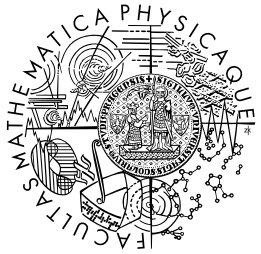

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

David Koňařík

Query langauge for relational databases

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Tomáš Petříček, Ph.D.

Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

To everyone who didn't give up on my rocky studies and gave me a chance to get to where I am, and to Tomáš Petříček, my helpful supervisor.

Title: Query langauge for relational databases

Author: David Koňařík

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D., Department of Distributed and Dependable Systems

Abstract: Relational databases are widely used in software engineering, but the language used to access them, SQL, was created in the 1970s and has numerous problems that make writing complex queries unnecessarily difficult. In this thesis we will present some flaws of SQL, go through select existing alternatives, and define a new query language, PPPQL, based on the concept of a query pipeline. PPPQL improves on SQL by having a consistent syntax and semantics, allowing complex queries to be expressed more simply.

We will first introduce PPPQL through a series of examples, then define the syntax and semantics of the language, using a formal description for part of the language.

To validate PPPQL's design, the language was implemented as an extension to Postgres. We will discuss the most interesting parts of its implementation and see how an alternative query language can be implemented in Postgres.

Keywords: query language, relational database, language design, SQL, Postgres, PostgreSQL

Název práce: Dotazovací jazyk pro relační databáze

Autor: David Koňařík

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Relační databáze jsou široce rozšířené v softwarovém inženýrství, ale jazyk, kterým se k nim přistupuje, SQL, byl vytvořen v 70. letech minulého století a má řadu problémů, které komplikují složitější dotazy. V této práci poukážeme na některé vady SQL, projdeme vybrané existující alternativy a definujeme nový dotazovací jazyk, PPPQL, na základě konceptu posloupnosti transformací. PPPQL má oproti SQL konzistentní syntax a sémantiku, díky čemuž umožňuje jednodušeji vyjádřit komplexní dotazy.

Nejprve uvedeme PPPQL skrz řadu příkladů, pak definujeme syntax a sémantiku jazyka, včetně formálního popisu pro část jazyka.

Abychom validovali jeho design, jazyk PPPQL byl implementován jako rozšíření do Postgres. Projdeme nejzajímamější části jeho implementace a ukážeme, jak může být alternativní dotazovací jazyk implementován v systému Postgres.

Klíčová slova: dotazovací jazyk, relační databáze, návrh jazyků, SQL, Postgres, PostgreSQL

Contents

1 Introduction **6**

2 Existing query languages **8**
- 2.1 SQL . . . 8
- 2.2 DSLs in general-purpose languages . . . 10
- 2.3 PRQL . . . 11
- 2.4 Malloy . . . 11
- 2.5 SaneQL . . . 11

3 Introduction to PPPQL **12**

4 Syntax **14**
- 4.1 Lexer . . . 15
- 4.2 Literals . . . 15
- 4.3 Expressions . . . 16
- 4.4 Queries . . . 16

5 Semantics **17**
- 5.1 Expressions . . . 17
- 5.2 Queries . . . 18

6 Compiler implementation **21**
- 6.1 Parser . . . 21
- 6.2 Postgres representation of queries . . . 22
- 6.3 Analyser . . . 24
- 6.4 Expression translator . . . 24
- 6.5 Query translator . . . 25

7 Future improvements **27**

8 Conclusion **28**

Bibliography **29**

A Attachments **30**

B Installation and usage **31**

1 Introduction

Modern relational database management systems (RDBMSs) allow programmers to query them using a "query language", which declaratively specifies the steps to be taken to obtain the requested data. The RDBMS then "plans" the query, using statistical knowledge of the database to create an efficient "execution plan" that is then interpreted to provide the requested result.

All popular RDBMSs use SQL (created in the 1970s) as their query language, and there has been little focus on improving the basics of SQL, with new versions simply layering features onto the existing base by means of syntax extensions.

Although SQL has been successfully used in the field for decades, we consider SQL to be a flawed language, lacking multiple key qualities (see section 2.1):

- **Composability**: It's desirable for the language to allow any non-trivial piece of code to be factored out into its own reusable unit. In SQL, this can be achieved in queries only by "Common Table Expressions" at the table-level, or by functions outside the query. CTEs, however, can't be parametrised. Functions are unwieldy and don't allow e.g. easily passing aggregate columns or returning multiple columns.

- **Orthogonality**: SQL contains multiple concepts that can serve the same function, but can't be used interchangeably in its constructs. For example, aggregate columns may only be used as arguments to aggregate functions. Consequently, they can not be passed to regular functions as sets of values. Tables can be used as arrays of records, but this requires a specific query to accomplish. While queries can be nested, this won't produce a nested structure on the result, despite this being often desirable in application programming.

- **Consistency**: In SQL, queries are written in a rigid form in a `SELECT` statement, which is specified as a single syntactic block with fixed clauses that modify the query. These clauses, however, semantically execute in a different order to their syntactic order. This is unintuitive, and forces programmers who require a different order to resort to nested queries, instead of simply reordering the clauses.

  Furthermore, features that would be implemented as functions in other languages are often implemented with dedicated syntax and keywords, with little thought given to consistency.

There have been attempts to create a new query language for RDBMSs (discussed in chapter 2). These projects have mostly been implemented using translation to SQL, keeping it as the only language that RDBMSs themselves accept. This limits the options of the language's semantics, since they have to map to SQL source code, often without any prior knowledge of the underlying database.

We build upon those projects and design a new language, PPPQL (see chapters 4 to 5), with the goal of addressing SQL's issues outlined above. PPPQL is designed to be independent of SQL and implementable on any popular RDBMS. To give an example, consider this PPPQL query and its equivalent in SQL:

```
| from m := measurements
| limit 1000
| group s := sensor
| select s, m.temperature
           + 273.15
```

```
SELECT sensor,
       array_agg(temperature
                 + 273.15)
FROM (SELECT *
      FROM measurements
      LIMIT 1000) AS m
GROUP BY sensor
```

Note that in PPQL, the order of filters (`from`, `limit`, …) determines the semantics of the query, allowing us to simply use `limit` before `group`. Furthermore, the `group` filter produces a standard value of a set type, not requiring a special notion of aggregate columns as in SQL. For convenience, values in sets can be manipulated element-wise by all operators and functions, such as the `+` above.

We also introduce an implementation of PPPQL built as an extension to PostgreSQL [1] (also known as Postgres), an open source RDBMS. We go through the structure of its code and how it compiles PPPQL to Postgres' internal structures (see chapter 6). The code is available as an attachment to this thesis (see appendix A) and can be installed on a patched version of Postgres (see appendix B).

2 Existing query languages

SQL traces its history back to 1974 and was first standardised by ANSI in 1986. Unlike most other languages of that age, SQL is still the dominant language in its application domain, but there have been attempts to create alternatives. None of them have been successful enough to be included in a popular RDBMS, but they have many interesting ideas nonetheless.

In this chapter we will look at SQL, examine its flaws, and look at its various alternatives.

2.1 SQL

SQL, or Structured Query Language, is the most widely used standalone query language [2]. It's used not just in traditional RDBMSs, but also search engines (Manticore Search [3]) and time-series databases (QuestDB [4]).

Despite its popularity, SQL has many flaws from a programming language perspective. To give some rationale for the development of PPPQL and other alternatives, we will list some of them here.

It's important to note that, despite being standardised by ISO, SQL differs significantly from implementation to implementation. We will be looking specifically at the Postgres dialect of SQL.

Order of operations SQL queries have a rigid syntactical and semantic structure, requiring queries that need a different order of operations to use nested queries, which decreases readability. The static structure also has multiple elements which are functionally equivalent (`HAVING` vs `WHERE`), differing only in the semantic order. Moreover, the syntactic order is different from the semantic order, leading to confusion (see fig. 2.1).

For example, `SELECT` specifies the columns of the resulting table, executing last in the semantic order, but is written at the start of the query, hurting consistency.

```
SELECT sensor, avg(temperature)
-- A subquery must be used, since otherwise LIMIT would apply
-- after the GROUP BY
FROM (SELECT * FROM measurements
      ORDER BY timestamp
      LIMIT 1000)
GROUP BY sensor
```

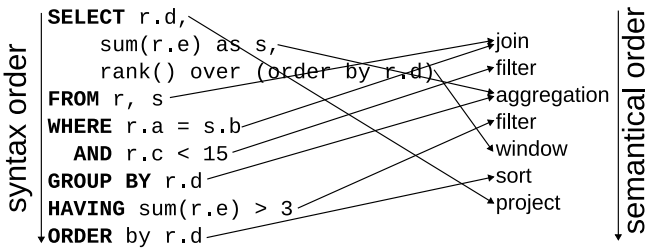

Figure 2.1 Diagram of the semantic order of an SQL query [5, Figure 1]

Sets of values SQL contains multiple concepts that group multiple values together: arrays, aggregate columns, JSON types. These are mutually incompatible and need to be converted, often in convoluted ways. Mapping over the contained values also often means writing a verbose subquery. This demonstrates SQL's poor orthogonality. For example, you might expect this query to be valid and produce a set of values:

```
SELECT name -- Error: can't select bare aggregate column
FROM people
GROUP BY division
```

In fact, to receive a list of all names, `name` must be wrapped in `array_agg()` to convert it to a value:

```
SELECT array_agg(name)
FROM people
GROUP BY division
```

Let's consider a table of articles, where each has a name and an array of tags. To convert those tags to uppercase, one might intuitively call `upper()` on `tags`, or try to use some `map` function:

```
SELECT name, upper(tags)
FROM articles
```

In SQL, the above would result in an error, since the `upper()` function only operates on individual strings, not arrays. we need a verbose nested query that treats the array as a table via `unnest()`:

```
SELECT name, (SELECT upper(tag) FROM unnest(tags) AS tag)
FROM articles
```

Abstraction mechanisms SQL queries offer only very limited means of abstraction: Common Table Expressions (CTE) and functions (defined outside queries). Furthermore, abstracting over both aggregate columns and arrays is impossible directly and requires overly verbose conversions. This discourages composing complex statements from several smaller, simpler pieces, a common practice in most programming languages.

The following code creates and uses a function that computes multiple statistical functions for a specific set of values. It receives the values as an array and returns the results as a JSON object, requiring the query using it to perform conversions:

```
CREATE FUNCTION stats(measurements decimal[])
RETURNING json LANGUAGE SQL AS $$
    SELECT json_build_object(
        'avg', (SELECT avg(x) FROM unnest(measurements) AS x),
        'min', (SELECT min(x) FROM unnest(measurements) AS x),
        'max', (SELECT max(x) FROM unnest(measurements) AS x));
$$;
```

```
SELECT
    stats(array_agg(temperature)) AS temperature_stats,
    stats(array_agg(humidity)) AS humidity_stats
FROM measurements
GROUP BY timestamp::date
```

Inconsistent features Any semantic model of SQL is complicated by obscure and inconsistent features that don't follow simple intuition. For example, Postgres contains "set-returning functions", which can be used in place of expressions, but actually modify the containing query:

```
-- These two queries both create a table of 5 rows
SELECT x FROM generate_series(1, 5) as x;
SELECT generate_series(1, 5) AS x;
```

The `SELECT` clause is followed by a list of expressions and one would expect each expression to have only a local effect, determining only its own value. In fact, `generate_series()` is a set-returning function, so it effectively adds a new source table when used as an expression.

Linked table access and NULL Relational databases often contain many tables, linked with referencing keys (SQL `FOREIGN KEY` constraint). Accessing these linked tables in queries necessitates adding a `JOIN` and manually restating that constraint to link relevant rows. SQL provides no short-hand for doing these traversals.

SQL handles NULL values akin to NaN values in other languages: a NULL value is not equal to another NULL value. This doesn't match other frequently used languages and is a frequent source of issues for new users.

2.2 DSLs in general-purpose languages

While SQL is the dominant language used by RDBMSs themselves, it's often used just as a protocol or intermediate representation for sending queries between processes. In such cases, SQL queries are generated by a library such as SQLAlchemy [6], jOOQ [7] and Entity Framework [8].

These libraries offer embedded DSLs to build database queries; the host language then serves as a meta-language for their construction. Their advantage is this integration with the host language, but they also fix some of SQL's design flaws: Queries can be created outside SQL's rigid order of operations, composition is handled by a potent host language, the syntax is simpler and queries can be introspected by looking at the "query object" generated.

C# is especially interesting here, since it offers an embedded "mini-language" in the form of LINQ, which translates to chained method calls.

Transpiling to SQL has its limitations and problems, however. Different RDBMSs have different dialects of SQL, necessitating some kind of compatibility layer. Since the dialects don't add features in a principled manner, supporting each new feature complicates the transpiler core further.

Running outside of the database also means that the compiler can't readily access database metadata, putting it at a disadvantage to SQL or requiring a build/code generation step to pull the metadata, which is required to be static, beforehand.

2.3 PRQL

PRQL [9] is perhaps the best known standalone SQL replacement language. Like most others, it models queries as pipelines of data transformations, letting the user compose query operations in whichever order they need. The canonical implementation compiles PRQL to SQL.

Instead of reusing the database's functions, it uses its own standard library and allows user-declared functions. Use of existing functions in the database is only possible through "s-strings", templated pieces of SQL. This improves composability inside the language, but means that use of many RDBMS features relies on user-defined string concatenation templates.

2.4 Malloy

Malloy [10] is another language designed to compile into SQL. It is mainly aimed at analytical queries. It integrates with a GUI dashboard and encourages users to write abstractions over existing data sources, adding "measures" (calculated aggregates) and "dimensions" (calculated scalar values) and derived tables.

Malloy's queries are composed of "stages", which are mostly modelled on SQL's SELECT statements. Like SQL, the stages consist of parts whose semantic order of execution is hardcoded and may not match the syntactic order.

2.5 SaneQL

SaneQL [5] was introduced in a recent paper critiquing SQL as a simple alternative. It uses pipeline semantics with operators, allowing users to define new ones in a macro-like fashion, improving on SQL in both syntax and abstraction mechanisms. Its reference implementation is a transpiler that creates SQL code.

Of the languages listed here, it is the most recent (published for the 2024 Conference on Innovative Data Systems Research) and most similar to PPPQL. Its `group` operation, however, seems to be much less general, requiring all resulting columns to be explicitly named and created by aggregate functions.

3 Introduction to PPPQL

Before we show a more complete description of PPPQL, let's see how to use the language on Postgres from a user's perspective and see how it improves upon SQL.

Once the extension is installed (see appendix B), PPPQL queries can be submitted just like normal SQL queries (by `psql` or any other client), they just need to be prefixed with `PQL`.

A PPPQL query is a sequence of filters. Most queries start with the `from` filter, which creates a query from the contents of a table:

```
| from buildings
```

The `from` filter introduces a variable named after the table, effectively "copying" the table's data into the query. To access the table's columns, we can either use the field access operator (a dot), or use just the column name, since PPPQL will implicitly search all row variables for brevity.

```
| from buildings
| select buildings.faculty, city, name
```

Further filters can be added to this query. They mostly behave similarly to parts of SQL queries of the same name. The query below filters on a predicate (`faculty = "MFF"`), groups the rows by city and returns a table of two rows: a city and an array of the names of its buildings:

```
| from buildings
| where faculty = "MFF"
| group c := city
| select c, buildings.name
```

Note that the syntactic order of filers exactly matches their semantic order of execution, unlike in SQL queries.

New variables can be introduced with `let`, improving on SQL's composability. Note the use of `||`, a native Postgres operator for string concatenation:

```
| from buildings
| let label := name || " (" || faculty || ")"
| select label
```

Since most elements of a query are expressions, and a subquery itself is an expression, PPPQL is more orthogonal than SQL and any abstraction mechanisms are more powerful. In the above queries, `buildings`, `faculty = "MFF"`, `c` are all expressions and can be used (mostly) interchangeably.

We can use the database's existing functions and operators in PPPQL, with the added bonus that operations are automatically lifted to apply elementwise on sets. Boolean operators are written as words with a `--` prefix, to disambiguate them without the need for keywords.

The following query will take all temperature measurements from specific sensors and return one row per day, with a set of temperatures:

```
| from measurements
| where sensor = "outside"
   --or sensor = "inside"
| group day := date(timestamp)
| select day, measurements.temperature + 273.15
```

4 Syntax

To start the description of PPPQL, let's look at a formal description of the syntax. The goal is to sufficiently describe PPPQL to its users and to allow the creation of an independent implementation.

PPPQL's syntax was chosen to be simple to parse and to generate, while also being friendly for programmers to write and read. Unlike SQL's syntax, it reflects the semantic model of the PPPQL compiler by modeling queries as sequential pipelines.

The syntax is described here in a variant of Backus–Naur form (BNF), with some elements added to the base:

- String literals can contain escape sequences with the same meaning as in C.
- `<any>` matches any character (needed for parsing string literals).
- `<empty>` matches an empty string.
- Character ranges like `"a"..."z"` match any character within a class (here lowercase letters).
- Parentheses (`(`, `)`) group expressions for use with operators
- The infix minus operator (e.g. `<left> - <right>`) matches anything that is matched by its left operand while not matched by its right.
- The postfix question mark operator (e.g. `<expr>?`) matches either its operand expression or an empty string. It can be translated as `(<expr> | <empty>)`.
- The postfix asterisk operator (e.g. `<expr>*`) matches any number of repetitions of its operand expression, including zero matches.
- The postfix plus operator (e.g. `<expr>+`) matches any non-zero number of repetitions of its operand expression.
- The postfix comma plus operator (e.g. `<expr>,+`) matches a comma-separated list of at least one repetition of its operand expression.

4.1 Lexer

The PPPQL compiler's parser does not have a separate lexer, but it does follow some simple tokenisation rules for consistency.

```
<whitespace_char>  ::= " " | "\t" | "\n" | "\r"
<digit>            ::= "0"..."9"
<ident_start_char> ::= "a"..."z" | "A"..."Z" | "_"
<ident_char>       ::= <ident_start_char> | <digit> | "$"
<operator_char>    ::= "+" | "-" | "*" | "/" | "<" | ">" | "=" | "~"
                     | "!" | "@" | "#" | "%" | "^" | "&" | "|" | "`"
                     | "?" | "." | ":"

<comment>  ::= "--#" (<any> - "\n")* "\n"
<ws>       ::= (<whitespace_char> | <comment>)*
<ident>    ::= <ident_start_char> <ident_char>* <ws>
<operator> ::= (<operator_char>* | "--" <ident_char>*)
             - (":" | "|") <ws>
```

Tokens can generally be followed by any amount of whitespace (space, tab, newline characters or line comments). Identifiers are composed of a starting character (letters of the English alphabet and underscore, excluding numerals to prevent ambiguity) and any number of subsequent characters (adding digits and the dollar sign).

Operators are generally composed of the characters seen above, but there are further rules forbidding some combinations (see below). Operators can also be alphanumerical if prefixed with `--`.

4.2 Literals

```
<string_literal>  ::= '"' (<any> - '"') '"' <ws>
<decimal_literal> ::= <digit>+ ("." <digit>*)? <ws>
                    | "." <digit>+ <ws>
```

String literals are put between double quotes, without any escape sequence recognition. Decimal numbers can be written without a decimal point for integers, with a decimal point and no following digits, with a decimal point and no leading digits, or in full with a decimal point followed and preceded by digits.

For example, `"abcd"` is a string literal, and `1.2`, `.2` and `1.` are all decimal literals.

4.3 Expressions

```
<argtuple_flag>   ::= ":" <ident> <ws>
<argtuple_kwarg>  ::= ":" <ident> <ws> <expr>
<argtuple>
    ::= (<argtuple_kwarg> | <argtuple_flag> | <expr>)
        (<argtuple_kwarg> | <argtuple_flag> | "," <ws> <expr>)*
      | <empty>
<argtuple_literal> ::= "[" <ws> <argtuple> "]" <ws>
<function_call>    ::= <ident> "(" <ws> <argtuple> ")" <ws>
<subquery>         ::= "(" <ws> <query> ")" <ws>
<parens>           ::= "(" <ws> <expr> ")" <ws>
<prefix_op>        ::= (<operator> - ".") <expr>
<infix_op>         ::= <expr> (<operator> - ("." | ":=")) <expr>
                     | <expr> "." <ident> | <ident> ":=" <expr>
<expr>             ::= <string_literal> | <subquery> | <parens>
                     | <decimal_literal> | <argtuple_literal>
                     | <function_call> | <ident> | <prefix_op>
                     | <infix_op>
```

PPPQL uses "argtuples" in multiple places as arguments to functions or query filters, and as literals for arrays and dictionaries. They hold values both by index ("positional arguments") and by name ("keyword arguments"). Lone "keywords" are treated as flags – keyword arguments with an implicit value of `true`.

For example, `score :desc, name :asc` is an argtuple that might be used to specify an ordering. It contains two positional arguments (`score`, and `name`) and two flags (`:desc`, `:asc`). A hypothetical function might take named arguments via argtuples like this: `percentile(data, :frac 95)`.

Argtuples may also be used as literals. Their use is implementation-specific and determined by the underlying database's capabilities.

Function calls follow the typical syntax, with arguments in the form of an argtuple enclosed in parentheses. Queries can be used as expressions, but need to be contained in parentheses to prevent ambiguity.

Infix operations are always left-associative, with a hardcoded table of priorities: `.`, `::`, `^`, `*`, `/`, `%`, `+`, `-`, default, `<`, `>`, `<=`, `=`, `=`, `!=`, `<>`, `&&`, `||`, alphanumeric, `:=` (leftmost operator binds tightest).

The field access operator, `.`, may only be used with an identifier as its right operand, and the assignment operator, `:=`, may only be used in specific contexts and is similarly restricted to only identifiers as its left operands.

4.4 Queries

```
<query_filter> ::= "|" <ws> <name> <argtuple>
<query>        ::= <query_filter>+
```

Queries are composed of one or more filters, which are separated by pipe characters, `|`. The whole query is prefixed by `|` to prevent ambiguity (e.g. in subqueries).

The top-level syntactic element may either be a query or a bare expression.

5 Semantics

PPPQL has two interconnected "modes" of operation: expression composition and query composition.

In expression composition, users start with literals and global variables (e.g. database tables), and build more complex expressions with operators and functions.

In query composition, users begin with a "filter" that provides data and apply further filters which filter and transform this data. At each step, the query represents the transforms necessary to get the desired data, and can be evaluated to get the data as an expression.

5.1 Expressions

Each expression has a known type, and those types are divided into two kinds: scalars and sets. Scalars are provided by the underlying database and comprise basic literals (strings, integers, floats…) and more advanced composite types (dates, records…). Sets are ordered collections of other sets or scalar types.

Scalar types may have fields (pairs of string name and type), which can be accessed with the field access operator, written . (dot). This is how rows of tables (records) are represented.

Function lifting Function calls and infix/prefix operations on scalars are automatically lifted to work on sets, performing the operation elementwise and returning a set. The function may also be "partially lifted", where some arguments are iterated over and others are used verbatim.

If such a lifted function is passed sets of different lengths, the result will have the length of the largest set and the input sets will be padded to this length by `NULL` values.

Evaluation Expressions generally function according to common programming language conventions. For completeness, their behaviour is briefly described:

- Literals (`<string_literal> | <decimal_literal> | <argtuple_literal>`) directly evaluate to the value they represent. Argtuple literals are represented as arrays. Due to limitations of Postgres, elements of such a literal must all have the same type and must all be scalars.

- Identifiers (`<ident>`) reference variables in the current scope or global object from the underlying database (tables and constants for PostgreSQL).

  For convenience, variable lookups look inside variables representing table rows, so table columns can be referenced without explicit field access.

- Function calls (`<function_call>`) evaluate to the value returned by the function. Functions are considered external to PPPQL, provided by the underlying database engine and identified by name.

- Prefix and infix operations (`<prefix_op> | <infix_op>`) behave like function calls, calling external code.

- Subqueries (`<subquery>`) are composed using the query logic described below and evaluated into a set of records. The query's filers inherit the subquery expression's variables.

5.2 Queries

PPPQL queries are a sequence of "filters", which are named functions that take the previous "variable table" (an imaginary table where every columns is a variable) and "query description" (an implementation-specific opaque value that describes a database query producing that variable table) and an argtuple of arguments, and produce a modified variable table and query description.

When a query is used as the top-level expression, its result is a set of records, one for each row of the variable table produced by the last filter. The representation of this set depends on the database used.

Certain filters can be used at the start of queries, where no prior variable table and query description are received. Others can be used only placed after a previous filter.

Filters generally interpret arguments as expressions (see above), but may use them directly as syntax trees for e.g. assignments and binary flags. Expressions in filters are evaluated in the context of a single row of the variable table.

For a semi-formal description of the filters' transformation of the variable table, we will use the following definitions:

- The universe U is the set of all values, E is the set of all untyped expressions (parse trees).
- The set of all rows with variables N (set of names) is denoted R_N.

  $R_N = \{(n, u) | n \in N, u \in U\}$ s.t. $\forall r \in R_N \; \forall n \in N \; \exists! \, u \in U \colon (n, u) \in r$. Verbally, a row is a set of name-value tuples where each variable has exactly one value (exactly one tuple).

  A row may also be considered as a function $r : N \to U$, evaluation corresponding to field access.
- A table T_N is an ordered multiset of values from R_N.
- Query filters are functions that transform a set of rows T_N and arguments (as parse tree expressions, specified by the filter syntax) into a new set of rows.
- Expression can be turned into values with $\text{eval}_C : R \times E \to U$, which evaluates an expression in the context of a given row according to the rules described in the previous section.

  Evaluation uses an opaque context C, which describes the host database environment, global variables, and variables provided from surrounding queries.

from starts a query with data from tables
Syntax: `"from" ((<ident:n_i> ":=")? <ident:t_i>),+`
Semantics: $\text{from}_C(n_1, t_1, \ldots, n_m, t_m) =$
$\{\{(n_1, r_1), \ldots, (n_m, r_m)\} \mid r_1, ..., r_m \in \text{eval}_C(\emptyset, t_1) \times \cdots \times \text{eval}_C(\emptyset, t_m)\}$.

The `from` filter can only appear at the start of a query. It creates a variable table as the Cartesian product of the rows of the given tables $(t_1, \cdots, t_m)$. It creates variables for each source row named after the table or named using the optional alias $(n_1, \cdots n_m)$.

select replaces columns with new ones computed from expressions
Syntax: `"select" ((<ident:n_i> ":=")? <expr:e_i>),+`

The behaviour of `select` depends on whether it is the first filter in a query.

If `select` is the first filter in query:
$\text{select}_C(n_1, e_1, \ldots, n_m, e_m) = \{\{(n_i, \text{eval}_C(\emptyset, e_i)) \mid i \in \{1, \ldots, m\}\}\}$,

If `select` is preceded by another filter in the query, with T being its result:
$\text{select}_C(T, n_1, e_1, \ldots, n_m, e_m) = \{\{(n_i, \text{eval}_C(r, e_i)) \mid i \in \{1, \ldots, m\}\} \mid r \in T\}$,

The `select` filter creates a new variable table with the same rows and only the evaluated expressions listed in its arguments $(e_1, \ldots, e_m)$ as columns. If an alias $(n_1, \ldots\ n_m)$ is not given, the new variable name is implementation-defined.

If used at the start of the query, it creates a single row with variables consisting of the expressions given evaluated against the global context.

where filters rows by a boolean expression
Syntax: `"where" <expr:c>`
Semantics: $\text{where}_C(T, c) = \{r \in T \mid \text{eval}_C(r, c) = \text{true}\}$

The `where` filter returns a query with the same variables, but keeping only those rows for which the condition evaluates to a truthy value (implementation-defined).

group partitions rows by keys and adds those keys as variables
Syntax: `"group" (<ident:n_i> ":=" <expr:k_i>),+`
Semantics: $\text{group}_C(T_N, n_1, k_1, \ldots, n_m, k_m) =$
$\{\{(n_i, u_i) \mid i \in \{1, \ldots, m\}\}$
$\cup \{(v, \{r(v) \mid r \in T_N \wedge k(r, \vec{k}) = \vec{u}\}) \mid v \in N\}$
$\mid u_1, \ldots, u_m \in U^m \wedge \exists r \in T_N \colon k(r, \vec{k}) = \vec{u}\}$,
where $k(r, \vec{k}) = (\text{eval}_C(r, k_1), \ldots, \text{eval}_C(r, k_m))$.

The `group` filter takes all existing variables and transforms them into sets of values, where each set consists of values from those rows, where `key` is constant. That value of `key` is then added as a new variable named per `key_name`.

order sorts rows by keys
Syntax: `"order" (<expr:e_i> (":asc"|":desc")?),+`
Semantics: $\text{order}_C(T, e_1, ..., e_m) = T'$ where T' is T sorted s.t. $s(T'_i) \leq s(T'_j)\ \forall i < j$ and $s(r) = (\text{eval}_C(r, e_1), \ldots, \text{eval}_C(r, e_m))$.

The i-th field of s has its order reversed if `:desc` is specified.

The `order` filter returns its input rows sorted according to the lexicographic ordering on the given expressions. The order is ascending by default, but can be changed for one expression by specifying `:desc`.

let adds a new variable
Syntax: `"let" (<ident:n_i> ":=" <expr:e_i>),+`
Semantics: $\text{let}_C(T, n_1, e_1, ..., n_m, e_m) =$
$T \cup \{\{(n_i, \text{eval}_C(r, e_i)) \mid i \in \{1, ..., m\}\} \mid r \in T\}$

The `let` filter adds new variables (i.e. columns) to the variable table.

limit returns the first n rows
Syntax: `"limit" <expr:n>`
Semantics: $\text{limit}_C(T, n) = \{T_1, \ldots, T_{n'}\}$, where $n' = \text{eval}_C(n)$

The `limit` filter removes any rows beyond the first n.

6 Compiler implementation

In this chapter we will look at the implementation of PPPQL on Postgres.

Since the compiler is implemented as a Postgres extension, it's first necessary to give an overview of how Postgres processes queries: When a query is received, its code is first parsed into a "parse tree" of nodes. This tree is then analysed into a "query tree", reusing some nodes from the parse tree and refining others. After analysis the rewriter applies rewriting rules, which e.g. implement views. The resulting query tree is then planned and executed.

To keep existing functionality working and retain as much of Postgres' performance characteristics, PPPQL replaces the topmost two stages: parsing and analysis. This means it has to produce the query tree, whose nodes mostly map to SQL concepts but afford some extra flexibility. Specifically, the output of the PPPQL compiler is the `Query` node (shown in fig. 6.2).

To allow keeping the PPPQL compiler separate from the larger Postgres codebase, it was implemented as an extension. Postgres already had a general extension mechanism, but there was no hook that allowed implementing a custom parser and/or analyser. To this end a hook was added to the existing "extensible node" mechanism, allowing extensions to register keywords, which, if detected at the start of a query, will bypass the standard Postgres parser and use an extension-specified function instead. A custom analyser can then be used for the resulting extensible node.

The PPPQL compiler is split into four stages: the parser, the analyser, the expression translator and the query translator. The parser takes the query as a string and creates a syntactical parse tree. The analyser then processes that tree by e.g. adding types and resolving identifiers, calling the query translator when it encounters a query. The query translator takes a query parse tree, processes each query filter, calling the analyser and expression translator in the process, and produces a Postgres `Query`. The expression translator takes the analyser's result and produces a Postgres `Expr`.

6.1 Parser

The PPPQL parser is implemented as a standard recursive-descent parser without a separate tokenisation stage. Since the PPPQL grammar is simple, the parser only consists of about 500 lines of code.

Possibly of note is its use of the PQLParseResult type and associated macros to simplify error handling and to somewhat emulate the style of parsers written in functional languages [11].

Every sub-parser can return one of three results: a success (`Some`), a miss (`None`) and an error (`Failed`). A set of macros helps with composing sub-parsers.

For illustration, consider the code snippet in fig. 6.1, which parses a function call. It first tries to parse an identifier and an open parenthesis. It uses the `TRY_SOFT` and `TRY_SOFT_REC` macros to return if either is not successfully parsed (e.g. if the source code contains a number instead). The later sub-parser calls are then wrapped in `TRY_HARD`, which likewise returns on a non-success, but considers even a miss to be an error.

```
static PQLParseResult parse_function_call(
        Parser *p, PQLParseTree *res) {
    Parser savestate = *p;
    PQLParseTree ident;
    TRY_SOFT(parse_ident(p, &ident));
    TRY_SOFT_REC(expect_str(p, "("), *p = savestate;);
    skip_whitespace(p);
    PQLArgTuple args;
    TRY_HARD(parse_argtuple(p, &args));
    TRY_HARD(expect_str(p, ")"));
    skip_whitespace(p);

    res->type = PQL_PT_Function_Call;
    res->function_call.function_name = ident.identifier;
    res->function_call.args = args;

    return PARSE_SOME;
}
```

Figure 6.1 An excerpt from the PPPQL parser source code

The sub-parser for expressions tries parsing all possibilities, trying the next variant if one returns a miss. If, however, one reports an error, the whole source code is considered erroneous and an appropriate error message is returned.

6.2 Postgres representation of queries

To understand how the analyser and translator work, it is necessary to know how Postgres represents the query tree. At the root of this tree is a `Query` node. This node is used for all SQL statements and has over 40 fields. Since PPPQL only produces queries, I will only describe the `SELECT` variant from now on.

Since `Query` is modelled after SQL, it has the same rigid semantic structure as an SQL query. It contains multiple phases roughly corresponding to the `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `DISTINCT` and `SELECT` parts of an SQL query. Queries that don't conform to this rigid structure need to be represented via workarounds, typically by nesting queries or using subqueries.

For example, to add a source table, one must add a `RangeTblEntry` to `rtable` and a `RangeTblRef` to `jointree`.

To group by an expression, it's necessary that this expression be present in the `targetList` (part of `SELECT` in SQL). Then a `SortGroupClause` is added referencing the operators required for comparing/hashing the expression.

When the query's rows are grouped by `GROUP BY`, the grouped columns conceptually contain sets of values, but in the query tree they behave as before, the only difference being that they can be used in aggregate functions.

To represent references to columns of tables, PostgreSQL uses a hybrid numbering systems: Columns are numbered from 1, with 0 representing the whole row. Tables inside one query are numbered from 1. The reference also needs to state which query the table is from (e.g. which `FROM`/`JOIN` clause introduced it); for this purpose a counter of "levels up" is used.

```
typedef struct Query
{
    NodeTag     type;
    CmdType     commandType;    /* select|insert|update|delete|merge|utility */

    bool        hasAggs;
    bool        hasWindowFuncs;
    bool        hasTargetSRFs;
    bool        hasSubLinks;
    bool        hasDistinctOn;
    bool        hasRecursive;
    bool        hasModifyingCTE;
    bool        hasForUpdate;
    bool        hasRowSecurity;
    bool        isReturn;

    List       *cteList;        /* WITH list (of CommonTableExpr's) */
    List       *rtable;         /* list of range table entries */

    List       *rteperminfos;   /* list of RTEPermissionInfo nodes */
    FromExpr   *jointree;       /* table join tree (FROM and WHERE clauses);
                                 * also USING clause for MERGE */

    List       *targetList;     /* target list (of TargetEntry) */

    List       *groupClause;    /* a list of SortGroupClause's */
    bool        groupDistinct;  /* is the group by clause distinct? */

    List       *groupingSets;   /* a list of GroupingSet's if present */

    Node       *havingQual;     /* qualifications applied to groups */

    List       *windowClause;   /* a list of WindowClause's */

    List       *distinctClause; /* a list of SortGroupClause's */

    List       *sortClause;     /* a list of SortGroupClause's */

    Node       *limitOffset;    /* # of result tuples to skip (int8 expr) */
    Node       *limitCount;     /* # of result tuples to return (int8 expr) */
    LimitOption limitOption;    /* limit type */

    List       *rowMarks;       /* a list of RowMarkClause's */

    Node       *setOperations;  /* set-operation tree if this is top level of
                                 * a UNION/INTERSECT/EXCEPT query */
} Query;
```

Figure 6.2 Abridged source code of the Postgres `Query` struct [12]

6.3 Analyser

The PPPQL analyser takes a `PQLParseTree` (an untyped syntax tree) from the parser and a variable context (a lookup table of strings to expressions) and returns a `PQLTExpr`, a typed expression tree. During this it resolves variables, looks up database objects like tables, functions and operators and generally notes all information required for translation of `PQLTExprs` into Postgres `Exprs`.

Each `PQLTExpr` has a type with one of two "kinds": scalar or set. Scalars can be any Postgres type except for arrays, and sets are any collections of scalars: arrays, results of queries or aggregate values.

Postgres has complex rules for resolving the correct function or operator to use on values of specific types, to e.g. ensure that addition of two integers results in another integer, even if floating point addition is also possible. The analyser defers to Postgres's code for finding the best match.

Infix operations with `:=` as the operator result in an error, since the assignment operator is not a valid expression. The operator is only used in query filters.

Subqueries are processed using the query filter subsystem (described later) and embedded in the `PQLTExpr` as a Postgres `Query`.

When the analyser encounters an identifier, it first tries to directly find a variable with that name, then to find a column in any `RowVar` of that name, then to find a table with that name.

6.4 Expression translator

The expression translator takes a `PQLTExpr` and converts it into a Postgres `Expr`, a recursive type used by Postgres for expressions.

PPPQL's use of a simple scalar-set dichotomy presents the biggest problems for translation of set expressions, since the translator has to consider that any set expression can be either an array, the result of a subquery (which can have one or many columns, further restricting its use), or an aggregate column. Only arrays are truly values and can be passed around as such, the other expressions must first be converted to arrays. Any operation on arrays beyond the existing functions, however, requires the use of a subquery with `unnest()`.

To illustrate the work the translator has to do, we'll give a few examples of queries in PPPQL and the results of their translation. For brevity, the results will be presented in SQL, even though the translator actually works on internal AST-like structures.

Consider a set variable named `temperatures`, which is represented by an aggregate column. When we evaluate this variable as an expression, it needs to be converted to an array, since SQL does not allow bare aggregate values, resulting in `array_agg(temperatures)`.

Let's say we want to add a constant `10` to this variable, which is now represented by an array. The translation of `temperatures + 10` requires the use of a subquery and a function call:

```
(SELECT array_agg(t + 10) FROM unnest(temperatures) AS t)
```

When a table is evaluated as an expression, the translator needs to create a subquery, so `measurements` is translated like this:

```
ARRAY(SELECT timestamp, sensor, temperature, humidity
      FROM measurements)
```

6.5 Query translator

The query translator takes a `PQLQuery` (a specific `PQLParseTree` node variant) and processes each query filter sequentially, building a Postgres `Query` in the process.

For simplicity, to avoid SQL's fixed query operation order, each filter (except for the first) wraps the `Query` produced by the previous filter, selecting from it as a subquery. This allows each filter to function without considering which filter preceded it.

Along with a `Query`, each filter produces a variable context – a hash table of variables that can be accessed inside that query, saved as PPPQL expressions (`TExprs`). This corresponds to the semantic variable table's columns. This context is then used to analyse and translate the expressions given as arguments to query filters.

Like with the expression translator, we will use some examples to illustrate the query translator's work.

The following PPPQL query is simple, but demonstrates how each query filter produces a standalone Postgres `Query`, requiring the use of information not obtainable by simple transpilers to SQL; here the column list needs to be obtained:

```
| from measurements
```

```
SELECT timestamp, sensor, temperature, humidity
FROM measurements
```

More complex PPPQL queries produce highly nested Postgres queries (see figure 6.3). Here we also see the expression translator at work, adding aggregate functions and subqueries where needed.

```
| from m := measurements
| group s := m.sensor
| select s, avg(m.temperature + 273.15)
```

```
SELECT
    s,
    (SELECT avg(u1) AS agg
     FROM unnest(ARRAY(
        SELECT u2 + 273.15 AS func
        FROM unnest(ARRAY(
            SELECT u3.temperature
            FROM unnest(i1.m) AS u3(...)))
        AS u2))
     AS u1)
FROM (
    SELECT
        array_agg(i1::record) AS m,
        i1.sensor AS s
    FROM (SELECT sensor, timestamp,
                 temperature, humidity
          FROM measurements) AS i1
    GROUP BY i1.sensor) AS i1
```

Figure 6.3 The output of translating PQL code (on the left) into SQL (on the right), edited for brevity

7 Future improvements

While PPPQL is a useable query language today, it doesn't allow users to fully utilise all of Postgres' features, and there are also features not present in SQL that could be added to it:

- **Functions or macros**: Allowing users to abstract repetitive parts of queries in a lightweight way (e.g. without requiring the creation of an extra database object) could simplify many complex queries. This abstraction would most likely take the form of functions (operating on values) or macros (operating on expressions). Functions are conceptually simpler and are used in modern programming languages; macros allow greater flexibility when e.g. abstracting over tables with similar columns.

- **Foreign key follow operator**: Normalised relational databases consist of many tables, often joined with simple one-to-one or many-to-one relationships. In PPPQL, like in SQL, they currently have to be traversed by manually adding the related table and restating the constraint.

  Thanks to the PPPQL compiler running inside the Postgres process, it can create virtual columns in tables for each `FOREIGN KEY` constraint, field access on which would then access the referred-to table.

- **More general container types**: Postgres contains arrays, but they can't be nested, only used as fixed-dimension matrices. This is limiting for PPPQL, since it can't translate e.g. queries involving multiple `group` filters.

  Postgres also doesn't have a heterogeneous array type or a dictionary type (apart from records and JSON types), limiting PPPQL argtuple literals and making complex query output harder. Regardless of whether one might want to store these types in tables, they would be useful for use with the functions described above and for letting applications get a multitude of information from a single query.

  Thankfully, Postgres allows the creation of new types by extensions, so these types could be introduced by PPPQL.

- **Queries on arbitrary sets**: Right now, queries have to either start with a `from`, which takes data from tables, or `select`, which creates a single row. This is mainly for ease of implementation, but there's no reason `from` couldn't operate on any sets of values in the future.

The Postgres extension could also be improved: Its resulting ASTs are currently unnecessarily verbose, some nesting could be removed to potentially improve performance.

It might be possible to upstream the minimal patch allowing the introduction of new languages into Postgres as extensions, making the use of PPPQL simpler and encouraging others to also experiment with new query languages.

8 Conclusion

In this thesis we have introduced the problem of designing a relational database query language, described issues in the currently dominant SQL and covered some other notable alternatives. We then described a new query language, PPPQL, and went through its implementation into Postgres.

We showed how PPPQL improves on SQL, noting that its syntax is more consistent and intuitive, that its simple scalar-set dichotomy improves orthogonality and that it generally provides more options for abstraction in queries.

As currently implemented, PPPQL is a useable query language covering basic relational operations. To make it better suited for complex scenarios, it could be extended by further filters and by integrating more host database functionality as functions, all while maintaining its basic structure as described in this thesis.

Running the compiler as part of Postgres itself and not compiling to SQL has allowed us to use the state of the database while compiling and to offer better error messages, so that users don't have to debug both PPPQL and SQL source code. This was also necessary to correctly process array columns in PPPQL. Furthermore, it gives us the option to implement more innovative features in the future, some of which were detailed in the previous chapter.

This approach has, however, complicated development somewhat, since it requires understanding Postgres' internals, instead of requiring only user-level knowledge of SQL.

We hope that this thesis contributes to efforts to design new and improved query languages and that further work is done on integrating these languages directly into RDBMSs, as we have done with PPPQL.

Bibliography

1. The PostgreSQL Global Development Group. *PostgreSQL* [online]. [visited on 2024-04-30]. Available from: https://www.postgresql.org/.
2. Stack Exchange Inc. *Stack Overflow Developer Survey 2023* [online]. 2023. [visited on 2024-04-30]. Available from: https://survey.stackoverflow.co/2023/.
3. Manticore Software Ltd. *Manticore Search Manual* [online]. [visited on 2024-05-08]. Available from: https://manual.manticoresearch.com/Introduction.
4. QuestDB Technology Inc. *QuestDB Query & SQL Overview* [online]. [visited on 2024-05-08]. Available from: https://questdb.io/docs/reference/sql/overview/.
5. Neumann, Thomas; Leis, Viktor. A Critique of Modern SQL And A Proposal Towards A Simple and Expressive Query Language. [N.d.]. Available also from: https://www.cidrdb.org/cidr2024/papers/p48-neumann.pdf.
6. SQLAlchemy authors and contributors. *SQLAlchemy 2.0 Documentation* [online]. [visited on 2024-04-30]. Available from: https://docs.sqlalchemy.org/en/20/.
7. Data Geekery GmbH. *The jOOQ User Manual* [online]. [visited on 2024-04-30]. Available from: https://www.jooq.org/doc/3.20/manual-single-page/.
8. Microsoft. *Entity Framework documentation hub* [online]. [visited on 2024-04-30]. Available from: https://learn.microsoft.com/en-us/ef/.
9. PRQL Developers. *PRQL language book* [online]. [visited on 2024-04-30]. Available from: https://prql-lang.org/book/.
10. Google. *Malloy Documentation* [online]. [visited on 2024-04-30]. Available from: https://docs.malloydata.dev/documentation/.
11. Leijen, Daan. Parsec, a fast combinator parser. [N.d.]. Available also from: http://users.cecs.anu.edu.au/~Clem.Baker-Finch/parsec.pdf.
12. The PostgreSQL Global Development Group. *PostgreSQL source file: parsenodes.h* [online]. [visited on 2024-04-30]. Available from: https://github.com/postgres/postgres/blob/master/src/include/nodes/parsenodes.h.

A Attachments

Two attachments are submitted as part of this thesis:

- `pg_patch.patch`, a patch to Postgres that allows the creation of new query languages via extensions.
- `pppql`, the source code of the PPPQL Postgres extension.

B Installation and usage

In this appendix, we will go through the process of building a patched version of Postgres and installing the PPPQL extension. This guide was tested on Debian 12, but will likely also work on other operating systems.

First we need to obtain the source code of Postgres, the patch enabling the creation of query language extensions, and the source code of the PPPQL extension. The last two are provided as attachments to this thesis, or can be obtained online[1].

We will use Postgres 16.2, the latest version available at the time of writing. With `postgresql-16.2`, `pppql`, and `pg_patch.patch` in the home directory, the following commands will build Postgres and install it in `~/pg`:

```
# Install build dependencies, run this as root!
apt install build-essential pkg-config libicu-dev \
            libreadline-dev zlib1g-dev
# Run the rest of these commands as a regular user
cd ~/postgresql-16.2
# Apply PPPQL patch
patch -p1 -i../pg_patch.patch
# Build and install Postgres
./configure --prefix=$HOME/pg
make
make install
# Allow running Postgres commands from PATH
export PATH=$HOME/pg/bin:$PATH
```

Now we need to build and install the PPPQL extension:

```
# Point Makefile to Postgres install
export PG_CONFIG=$(which pg_config)
cd ~/pppql
make
make install
```

We can now create and start a database that uses PPPQL:

```
# Create a new database
mkdir -p ~/db
initdb -E UTF8 -D ~/db
mkdir -p ~/db/sock
# Configure it to load the PPPQL extension
cat >> ~/db/postgresql.conf <<EOF
unix_socket_directories = '$HOME/db/sock'
listen_addresses = ''

shared_preload_libraries = 'pppql'
EOF
# Start the database system
pg_ctl -D ~/db -l ~/db/logfile start
# Create an empty database
createdb -h ~/db/sock $USER
```

[1] See git repository at https://gitlab.mff.cuni.cz/konarid/pppql

Now we can connect to the database with `psql -h ~/db/sock`.

To run a PPPQL query in `psql`, we prefix it with `PQL` and suffix it with a semicolon, like so: `PQL | select 1 + 1;`. We will receive output as if an SQL query had been sent instead.

Please note that `psql` interprets any occurrence of `--` as the start of a comment, so PPPQL queries containing this symbol will have to have their terminating semicolon on a separate line.