Freedom for the SQL-Lambda

Just-in-Time-Compiling User-Injected Functions in PostgreSQL

Maximilian E. Schüle
m.schuele@tum.de
Technical University of Munich

Jakob Huber
jakob.huber@tum.de
Technical University of Munich

Alfons Kemper
kemper@in.tum.de
Technical University of Munich

Thomas Neumann
neumann@in.tum.de
Technical University of Munich

ABSTRACT

As part of the code-generating database system HyPer, SQL lambda functions allow user-defined metrics to be injected into data mining operators during compile time. Since version 11, PostgreSQL has supported just-in-time compilation with LLVM for expression evaluation. This enables the concept of SQL lambda functions to be transferred to this open-source database system.

In this study, we extend PostgreSQL by adding two subquery types for lambda expressions that either pre-materialise the result or return a cursor to request tuples. We demonstrate the usage of these subquery types in conjunction with dedicated table functions for data mining algorithms such as PageRank, k-Means clustering and labelling. Furthermore, we allow four levels of optimisation for query execution, ranging from interpreted function calls to just-in-time-compiled execution. The latter—with some adjustments to the PostgreSQL's execution engine—transforms our lambda functions into real user-injected code.

In our evaluation with the LDBC social network benchmark for PageRank and the Chicago taxi data set for clustering, optimised lambda functions achieved comparable performance to hard-coded implementations and HyPer's data mining algorithms.

CCS CONCEPTS

• **Theory of computation** → Lambda calculus; • **Information systems** → **Structured Query Language**; *Main memory engines*; **Clustering**; **Query optimization**; **DBMS engine architectures**; • **Software and its engineering** → *Just-in-time compilers*.

ACM Reference Format:
Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *32nd International Conference on Scientific and Statistical Database Management (SSDBM 2020), July 7–9, 2020, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3400903.3400915

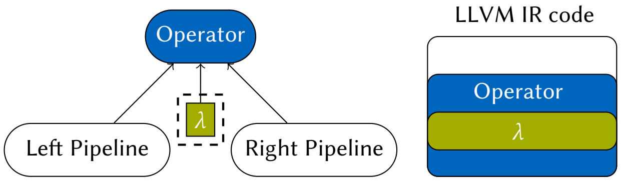


Figure 1: Operator tree with a lambda expression: code generation injects the lambda expression.

1 INTRODUCTION

Shifting computations to database systems is one of the main challenges that research on database systems is attempting to deal with [4, 11]. To shift the boundary between database systems and dedicated tools [5, 22], extensions such as MADlib [14] enable in-database analytics with dedicated table operators inside of relational database systems, but they still lack support for user-defined customisation.

For this reason, the main-memory database system HyPer [15] is equipped with lambda expressions for specifying metrics within data mining operators such as PageRank or k-Means. They allow user-defined distance metrics or node specification to be precompiled and injected into otherwise inflexible operators (see Figure 1). This is possible because HyPer generates code using the LLVM (low-level virtual machine) compiler backend. As research has until now been focused on HyPer, the publication of lambda functions has been focused solely on the specific case of a modern main-memory database system, while the corresponding source code is restricted.

To overcome these limitations, this paper shows how SQL lambda functions can be adapted to a conventional disk-based database system. In PostgreSQL, we extended the grammar to allow lambda expressions and subqueries as the table function's arguments. Table functions request tuples from subqueries or pre-materialise their results. In a similar way to code generation when evaluating expressions, we seamlessly injected lambda expressions into the table function's code. The specific contributions of this study are as follows:

- an extension of PostgreSQL's semantic analysis to support lambda functions as table function arguments,
- two different subquery types, a materialised table and an operator plan, either of which can be used inside of table functions,

SSDBM 2020, July 7–9, 2020, Vienna, Austria

ACM ISBN 978-1-4503-8814-6/20/07…$15.00
https://doi.org/10.1145/3400903.3400915

- a modification of PostgreSQL's just-in-time compiler framework to inline lambda expressions in table functions,
- their exemplary usage in combination with table functions for labelling, k-Means and PageRank,
- the corresponding source code published as open source,
- an evaluation in terms of scalability when varying the input size or the number of available threads.

This paper comprises the following sections: Section 2 describes the fundamentals of lambda expressions in general, including work related to lambda functions in HyPer and just-in-time compilation for PostgreSQL. Section 3 goes on to explain PostgreSQL database architecture with particular emphasis on PostgreSQL internals, such as developing extensions. Section 4 describes the high-level concept for integrating lambda expressions in each subpart of the PostgreSQL database system. Section 5 addresses some low-level C/C++ issues, particularly with regard to optimisation. Section 6 discusses the implementation of the three data mining algorithms, that will ultimately use the proposed lambda expressions. Section 7 evaluates the performance of the three data mining algorithms implemented in this study with different types of input data. The evaluation results will later be compared with an equivalent HyPer test run.

2 RELATED WORK

This section introduces just-in-time compilation for database systems and tools as used in data analysis. They form the foundation of lambda functions in the code-generating database system HyPer, whose table function's architecture is to be adapted for PostgreSQL.

2.1 Code Generation within Database Systems

PostgreSQL [26] is a disk-based relational database system with Volcano-style query execution [12], where the top-most operator requires the underlying ones to produce tuples. Code generation was introduced in HyPer [15] together with a bottom-up query execution model that pushes tuples upwards to the parent operator. Code generation avoids the overhead of interpreted function calls by first precompiling the query into the LLVM assembler.

Butterstein et. al. [6] demonstrated that expression evaluation in PostgreSQL is one of the most time-consuming processes in the database system for TPC-H queries and developed a separate evaluation library for code generation while leaving the rest of the PostgreSQL system unchanged. Melnik et. al. [17] subsequently demonstrated how the internal PostgreSQL functions can be compiled with LLVM to improve the execution speed by up to 20 %. Finally, in 2018, Andres Freund released LLVM support for expression evaluation as part of PostgreSQL version 11[1].

2.2 Data Processing Tools

Knowledge discovery on databases, which is usually done using dedicated tools such as Pytorch, Theano [3] or TensorFlow [1], involves data wrangling, data preprocessing, data analysis or even training of a model [13] to deploy neural networks [21]. Whereas the latter mostly relies on matrix algebra [18, 28], SQL with user-defined functions (UDFs) is particularly well suited to data preprocessing and data analysis [10, 23].

To avoid the overhead caused by user-defined functions [8], functions written in PostgreSQL's procedural language PL/pgSQL can be transformed into recursive SQL statements [9]. Instead of user-defined functions, hard-coded data-mining operators as provided by MADlib [14] for PostgreSQL or that form part of the main-memory database systems [25, 27] EmptyHeaded [2] and HyPer [15] achieve better performance for predefined data mining operators. We implemented similar table functions for exemplary usage in conjunction with lambda expressions and just-in-time (JIT) compilation.

2.3 Lambda Functions in SQL

Many programming languages offer lambda expressions that are anonymous functions and originate from the lambda calculus divised by Alonzo Church in 1936 [7]. As part of HyPer's extended SQL, lambda expressions allow users to inject customised code into otherwise inflexible table functions [20, 24]. The basic structure of an SQL lambda function in the HyPer database is as follows:

$$\lambda(name_1, name_2, ...)(expr). \tag{1}$$

For the remainder of this paper, the term *lambda arguments* refers to the named arguments between the first pair of parentheses, while the term *lambda expression body* refers to the expression between the second pair of parentheses. The lambda arguments refer to the tuple of a table that is passed as a subquery to the table function. The lambda expression body allows regular SQL expressions to be constructed out of the table's attributes. The following lambda expression calculates the L2 distance between two two-dimensional points given as part of S and T:

$$\lambda(S,T)((S.x-T.x)^2+(S.y-T.y)^2). \tag{2}$$

The lambda expressions are only valid when passed as a parameter in a call to an operator table function (such as k-Means), since semantic analysis and type inference is performed on the basis of the input row types to the function:

```
CREATE TABLE data(x float, y int);
CREATE TABLE centre(x float, y int); INSERT INTO ...
SELECT * FROM kmeans(
  (SELECT x,y FROM data), (SELECT x,y FROM centre),
  -- distance function and max. number of iterations
  λ(a,b) (a.x-b.x)^2+(a.y-b.y)^2, 3);
```

3 THE POSTGRESQL DATABASE SYSTEM

This section explains the components of the database system PostgreSQL, which have been modified to integrate lambda functions as subqueries and enable compiled execution later on.

3.1 Stages of Query Execution

PostgreSQL query execution comprises multiple stages. The raw query as a string is passed to the *parser*, which splits the query into tokens according to PostgreSQL grammar. The *analyser* then proceeds with the semantic analysis, which involves building expressions, resolving table names, attributes and function calls, and type checking. The modified query tree is then passed to the *planner/optimiser*, which outputs an optimised query tree. Finally, the plan tree is handed over to the *executor*, which performs the actual

[1] https://anarazel.de/talks/fosdem-2018-02-03/jit.pdf

table scans and function calls and evaluates expressions in order to produce the result set.

3.2 Functions and Table Functions

Inside of SQL queries, PostgreSQL supports function calls according to the SQL:2011[2] standard. These functions return any built-in type as well as entire tables, which are then referred to as *table functions*. Two composite types of table functions are possible:

- TABLE, which includes a fixed row-type definition,
- SETOF RECORD, which does not include a row-type definition in the function definition and is suitable for functions whose return type depends on the input data.

Table functions returning SETOF RECORD therefore require the caller to provide a row-type definition as a *column definition list*:

```
SELECT * FROM foo(1, 2) as (a int, b int);
```

Here, the table function foo returns a row type consisting of two integer values. The TABLE return type can be used if the row type of the function is guaranteed to always be the same. Then the fixed row type must already be provided already in the table definition, as follows:

```
CREATE FUNCTION foo(int, int) RETURNS TABLE (a int, b int) AS 'foo
      .so', 'foo' LANGUAGE 'C';
```

The function is now guaranteed to always return the type specified above and can be called, without providing a column definition list, as follows:

```
SELECT * FROM foo(1, 2);
```

PostgreSQL functions do not support multi-row arguments, i.e. no subquery returning more than one row can be passed as a parameter to a function. This poses a problem, which will be solved by providing tuple descriptors, as shown in the following sections.

3.3 Important Data Structures

Knowledge of the following PostgreSQL data structures is necessary to be able to follow the implementation details in this paper:

- A **tuplestore** is an internal data structure used for materialisation purposes, such as sorting tables or storing the tuples returned by a table function.
- **Datum** represents a single value in PostgreSQL, either a constant value or a pointer to a complex one, and is an alias to uintptr_t.
- **Object identifiers (Oid)** are integer values that are assigned for each PostgreSQL data type, table or function.

3.4 Extensions and the Function API

Extension functions have full access to all the internal PostgreSQL data structures, procedures and system catalogues and can therefore also be used to analyse and influence internal PostgreSQL processes. Each function must respect the PostgreSQL function API protocol for reading arguments and returning values. Most importantly, when returning a multi-row result from a set-returning function, the function must specify the return mode:

- SFRM_ValuePerCall specifies that the function expects to be called repeatedly, each time returning a single tuple or an end signal, and
- SFRM_Materialize specifies that the function has returned all the result tuples in a tuplestore (see Section 3.3) and it is not expected that it will be called again.

3.5 JIT Compilation

Just-in-time (JIT) compilation with LLVM enables the dynamic generation and compilation of code on-the-fly from a running program. Since version 11, PostgreSQL has supported JIT compilation for frequently used expressions, such as WHERE predicates, to improve the query execution performance. When compiling PostgreSQL, JIT compilation has to be enabled with a flag (--with-llvm).

The planner/optimiser will determine whether and to which types JIT compilation is applied. The PostgreSQL wrapper for LLVM offers multiple optimisation types, for example:

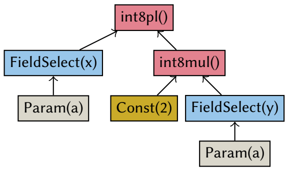
- PGJIT_EXPR enables the compilation of expressions,
- PGJIT_OPT3 enables strong O3 optimisation during compilation,
- PGJIT_INLINE inlines certain function calls in the expression based on a cost model, and
- PGJIT_DEFORM optimises column accesses by precomputing attribute offsets of common row types.

LLVM interacts well with the Clang C compiler, which can generate IR (Intermediate Representation) bitcode directly from the PostgreSQL C sources. This allows the PostgreSQL JIT system to consider its internal functions for inlining and optimisation without the need to reimplement them in LLVM IR code.

3.6 Expression Evaluation

Expressions in PostgreSQL queries are parsed to expression trees (sub-trees of the parse tree) and are evaluated in the executor stage. Each expression tree is first compiled to a sequence of opcodes, that represent elementary operations such as are needed to evaluate any expression. There are about 130 different opcodes in total, each representing one small step in an expression, such as loading a constant value, calling a function or selecting a specific attribute from a record value. The opcode sequence generated from the expression tree is the basis of interpreted and compiled evaluation.

```
SELECT a.x + 2 * a.y FROM ...
```



Step	Opcode	Comment
1.	EEOP_PARAM_EXTERN	a
2.	EEOP_FIELDSELECT	.x
3.	EEOP_CONST	2
4.	EEOP_PARAM_EXTERN	a
5.	EEOP_FIELDSELECT	.y
6.	EEOP_FUNCEXPR	int8mul()
7.	EEOP_FUNCEXPR	int8pl()
8.	EEOP_DONE	End

Figure 2: Example expression with corresponding parse tree and the opcode sequence generated from it.

[2] http://www.postgresql.org/docs/current/static/features.html

Interpreted evaluation simply loops over all steps and directly performs the computations. Compiled evaluation utilises JIT compilation based on LLVM. For this purpose, the compiling routine loops over the opcodes in a similar way as the interpreter, but rather than evaluating the opcodes immediately, the expression is rebuilt using LLVM primitives such as basic blocks, function calls and conditional jumps. To avoid unnecessary overhead, the compilation is delayed until the evaluation is triggered for the first time.

4 HIGH-LEVEL CONCEPT

The starting point for the implementation of lambda expressions is PostgreSQL version 11.2. This section describes the high-level concept of integrating lambda functions in PostgreSQL and consists of the lambda syntax, its evaluation and the integration as parameters in table functions.

4.1 Lambda Function Definitions

It is proposed, that the lambda functions for PostgreSQL will consist of named lambda arguments and an expression body, inspired by the HyPer syntax. Lambda expressions in SQL are treated as a parameter in a call to a table function.

The favoured approach is to make lambda expressions a dedicated language feature of PostgreSQL. As a dedicated feature, lambda expressions can be seamlessly integrated into the existing grammar and type inference system. This also makes the general PostgreSQL expression optimiser aware of the lambda expressions, especially when it comes to analysing and simplifying the lambda expression body.

Not just the expression body but the entire lambda function is treated as an expression, as it must be passed as a table function parameter. We now introduce a new pseudotype called LAMBDA, which is treated as an expression inside of a table function definition. The table function can access all information associated with a lambda expression in a newly defined data structure. This structure contains information about the named arguments, the return type, the expression tree of the actual expression and the row-type information for all the parameters involved.

4.2 Passing Input Data to Table Functions

Contrary to HyPer, PostgreSQL only allows subqueries as input for table functions that return a single tuple. Passing the table name as a string to the table function in a similar manner to MADlib requires another database connection to retrieve the data. However, this is unfeasible, since the semantic analysis of the lambda expressions cannot deduce the return type of lambda functions during compile time.

We therefore adjust the PostgreSQL subquery system to support multi-column and multi-row subqueries. We thus introduce two new pseudo data types, namely LAMBDACURSOR and LAMBDATABLE. They both indicate that a table function expects a subquery at the argument position, without any restrictions on row or attribute count. Figure 3 shows the difference between the two types.

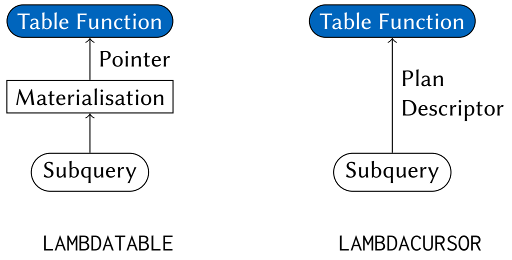


Figure 3: Comparison of LAMBDATABLE and LAMBDACURSOR modes for passing input data to table functions.

LAMBDATABLE fully materialises the subquery result into a tuplestore before passing it to the function. It is appropriate for table functions, which perform multiple iterations over the data and calculate the exact memory requirements based on the number of input tuples before processing the data. The table function receives a pointer to the tuplestore containing the materialised tuples when it is called.

LAMBDACURSOR does not materialise the subquery result. Rather than a tuplestore with all materialised tuples, the table function receives the subquery plan descriptor. This allows the table function to request the data tuple by tuple directly from the subplan. The table functions have no prior information as to the number of returned tuples.

4.3 Efficient Lambda Evaluation

As the injected lambda expressions, such as distance metrics, are core parts of the table functions, we extend the PostgreSQL executor to precompile these expressions. This paper introduces two optimisation levels of lambda evaluation in addition to interpreted execution and JIT optimisation. They apply to algorithms working with certain data types, e.g., float8 and int64:

- **Interpreted execution (L1)**: The lambda expression is evaluated as an ordinary PostgreSQL expression with a computed goto approach.
- **JIT-compiled execution (L2)**: The lambda expression is evaluated as a JIT-compiled expression, using existing JIT optimisations of the PostgreSQL executor.
- **High-performance JIT-compiled execution (L3)**: The lambda expression is rebuilt with a custom LLVM wrapper, using basic LLVM operators and native mathematical functions, which are optimised to generate highly efficient, compact code. Only a small subset of the PostgreSQL data types and arithmetic functions is supported.
- **High-performance JIT injection (L4)**: Same as the previous mode, but the code generated from the lambda expressions is directly injected into the table function or worker thread code, achieving a near hard-coded performance by completely optimising external function calls away.

4.4 Table Function Design

The data mining operators will be implemented as table functions, each bundled in a shared library. PostgreSQL automatically loads the library when accessing a specific table function for the first time. If the database has been compiled with `--with-llvm`, the PostgreSQL build system can be configured to use the Clang compiler to automatically generate LLVM IR bitcode files (`.bc`) from specific source files. In our case, the table functions were developed in C. This allows stronger JIT optimisations including lambda injection.

The bitcode can be loaded as an LLVM module which may be combined with the IR code generated from the lambda expressions. Most importantly, it is possible to inject the lambda expression bitcode into the table function bitcode, which will greatly improve the run time due to the reduced number of calls to the lambda evaluation function. Furthermore, it might be feasible to perform lambda injection not only for the table functions themselves but also for worker thread functions, which are executed in parallel.

4.5 Dynamic Row Types

The row format returned by a data mining table function usually depends on the input data. For example, a k-Means function produces the same data that it receives as input, plus one extra column containing an integer describing the assigned cluster number. When PostgreSQL cannot deduce the type automatically (as for `SETOF RECORD`), the SQL query must provide a column definition list. In the following, the table functions using lambda expressions will return `SETOF RECORD`, as the returned row format always depends on the input format.

5 LAMBDA INTEGRATION

This section explains the implementation of the lambda expressions in PostgreSQL, as provided online[3]. The main focus is on the necessary extensions required in the various execution stages of the PostgreSQL query.

5.1 Parser and Analyser Extensions

The extension for lambda expressions affects both parser and the analyser. We will first describe the PostgreSQL grammar for parsing lambda expressions and then introduce a new data structure that represents lambda expressions in a parse tree. Finally, we will describe the semantic analysis based on these building blocks.

5.1.1 Syntax Definition. The foundation for the lambda expressions is formed by a PostgreSQL grammar syntax extension. In a first step, the PostgreSQL parser is extended by a new grammar element inspired by the HyPer syntax described in Section 2.3 as follows:

$$\textbf{LAMBDA}(name_1, name_2, ...)(expr).$$

We prefer the LAMBDA keyword to the λ symbol as it avoids text encoding conflicts with non-ASCII characters, which would lead to problems with the PostgreSQL lexer. The $name_n$ $(n \geq 1)$ variables denote an arbitrary and unique identifier for the row values used in *expr*, which may be any regular PostgreSQL expression.

Given the lambda expression syntax defined above, it is not necessary to extend the PostgreSQL lexer, because there are no tokens that need to be introduced by PostgreSQL. Hence, all the necessary adjustments can be made in the actual grammar.

Inspection of the syntax defined above reveals that the lambda expression grammar consists of three parts, namely:

- the `LAMBDA` keyword,
- the comma-separated argument identifier list, and
- the expression body.

$$
\begin{array}{rcl}
\langle \text{lambda\_ident\_list} \rangle & \models & \langle \text{lambda\_ident\_list} \rangle \texttt{ , } \mid \langle \text{ColId} \rangle \\
\langle \text{lambda\_expr} \rangle & \models & \texttt{LAMBDA (} \langle \text{lambda\_ident\_list} \rangle \texttt{)(} \langle \text{a\_expr} \rangle \texttt{)} \\
\langle \text{a\_expr} \rangle & \models & \ldots \mid \langle \text{lambda\_expr} \rangle
\end{array}
$$

Figure 4: Grammar rules for parsing a lambda expression definition in Backus-Naur form.

Apart from being a keyword, `LAMBDA` should also denote a new built-in data type. In the lambda expression definitions, the `LAMBDA` keyword is followed by an opening parenthesis and therefore be mistaken for a call to a function named `LAMBDA`. To avoid this, the keyword must be defined as a `COL_NAME_KEYWORD`, which can be used as a column name but not as a function or type name.

The argument identifier list is simply a comma-separated enumeration of identifiers, which can be any string accepted as a column name. The expression body is expanded by the generic PostgreSQL arithmetic expression rule denoted by `a_expr`. Respecting these definitions yields the grammar rules for lambda expressions shown in Figure 4.

Since the `LAMBDA` keyword is defined as a `COL_NAME_KEYWORD`, it is possible for a `CREATE FUNCTION` statement to contain a `LAMBDA` function parameter, which can be parsed correctly as a type name without requiring any changes to the parser.

5.1.2 The LambdaExpr Node. The parsing stage transforms the input SQL query into a parse tree. For example, possible tree nodes exist for the `SELECT` statement, the `FROM` clause and the `WHERE` predicate.

It is also feasible to represent a lambda expression definition as an `Expr` node, since it is considered to be an expression according to the syntax definition. Furthermore, all of the information associated with a lambda expression, such as return type information or argument names, will be used in the parsing, analysing and execution stages. For this purpose, a new node named `LambdaExpr` will be introduced. This data structure will not only hold parsing information but is also passed to table functions, allowing them to perform additional type checks.

```c
typedef struct LambdaExpr {
  Expr  xpr;
  List  *args;       /* the arguments (list of row aliases) */
  Expr  *expr;       /* the lambda expression */
  List  *argtypes;   /* list describing the argument row types */
  Oid   rettype;     /* return type of the lambda expression */
  int   rettypmod;   /* return typmod of the lambda expression */
  Node  *exprstate;  /* ExprState for execution */
  Node  *econtext;   /* ExprContext for execution */
  Node  *parentPlan; /* parent PlanState */
  int   location;    /* token location, or -1 if unknown */
} LambdaExpr;
```

Figure 5: The definition of the LambdaExpr C structure.

Figure 5 shows the C definition of the lambda expression node. Only two of the fields are filled during the parsing stage. PostgreSQL builds the parse tree node by node. If the rule for a lambda expression defined in the previous subsection matches, a list is created for the argument names (stored in the `args` field) as well as an

[3] https://gitlab.db.in.tum.de/JakobHuber/postgres-lambda-diff

Expr node for the expression body. The location field is set to the position in the SQL query string where the match starts, which can be used to report exact error positions in the event of any syntax errors. These three fields are the only ones filled during parsing.

5.1.3 Type Definitions. Now that the grammar rules for lambda expressions have been defined, the next step is to register a new lambda type in the system catalogues. Since lambda expressions are intended to become a deeply integrated feature of the PostgreSQL database, the lambda type is registered as a new built-in type. All of the built-in PostgreSQL types (such as integer, text and double precision) are registered in the internal pg_type catalogue.

Each built-in type definition is stored in a data file (pg_type.dat) together with its name, a unique Oid, and multiple I/O functions. The I/O functions are Oids of PostgreSQL functions, which convert binary data to the specified type or perform type casts. In our case, the I/O functions are defined as dummy functions, which reject any attempt at converting expressions to or from lambda expressions. This makes the LAMBDA type a special sort of *pseudo-type*, in the sense that expressions of this type can only be read during table function evaluations. Such pseudotypes are also used in PostgreSQL for polymorphic types like ANYELEMENT or ANY.

In addition to the LAMBDA type, the two additional subquery types LAMBDATABLE and LAMBDACURSOR (as explained in Section 4.2) are registered as pseudo-types in the same fashion.

```
CREATE OR REPLACE FUNCTION foo (LAMBDATABLE left, LAMBDACURSOR
      right, LAMBDA expr) RETURNS SETOF RECORD AS 'bar.so', 'foo'
      LANGUAGE 'C';
```

Figure 6: A statement for creating a table function using the new pseudo-types.

These three new types having been registered, it is now possible to specify LAMBDA and the two subquery types LAMBDACURSOR and LAMBDATABLE respectively as parameter types and as part of a CREATE FUNCTION statement, as shown in Figure 6. In this example, a table function named foo is defined, with two subquery parameters (one a LAMBDATABLE and the other a LAMBDACURSOR) and one lambda expression. The types are generic and do not define any kind of restriction with regard to the type returned by the lambda expression, the number of lambda arguments or number/types of columns returned by one of the subqueries.

5.1.4 Semantic Analysis. Now that the syntax extension and type definitions have been prepared, we can extend the semantic analysis as part of the PostgreSQL analyser. The purpose of the analyser is to perform type deductions, check for semantic or other errors that cannot be detected by a simple syntax check, and resolve the tables, function calls and other variables used in the query. In particular, the adjustments necessary for integrating lambda expressions involve subquery analysis and lambda argument type deduction.

5.2 Planner Adjustments

The PostgreSQL planner transforms the query tree, returned by the analyser, to a query plan tree. The nodes of a query plan tree represent operations such as table scans, index scans, function scans and joins. By default, subqueries returning multiple rows (such as in an ARRAY expression) are implemented as a subquery scan followed by a materialisation step. The planner distinguishes between the two new lambda subquery types: the LAMBDATABLE type requires an additional materialisation step, which is not needed by the LAMBDACURSOR type.

5.3 Executor Stage

The executor must be adjusted to respect lambda functions as expressions, which can be passed as parameters to table functions. Furthermore, the executor needs to provide an interface, which allows the table functions to efficiently evaluate the lambda functions as needed.

5.3.1 Subquery Execution. The LAMBDACURSOR and LAMBDATABLE subquery types now need to be considered in the executor stage. The existing executor routines for subqueries first execute the subquery and then process the results according to the type of the subquery, such as reading the returned rows into an array or evaluating an EXISTS expression. The implementation of the two new types will follow this pattern.

The LAMBDATABLE implementation is constructed in a similar way to the ARRAY subquery evaluation, for which the executor initialises an empty array and reads all values produced by the (single-column) subquery into it. Likewise, for LAMBDATABLE, a tuplestore (see Section 3.3) is initialised and subsequently filled with the tuples (full materialisation).

The LAMBDACURSOR implementation omits the materialisation step. Instead, a pointer to the raw PlanState of the subquery is returned as a Datum pointer. This effectively delays the subquery execution until the execution of the table function.

5.3.2 Lambda Expression Initialisation. When traversing the query tree, the executor will process the lambda functions while preparing the arguments for the table function call. Each PostgreSQL expression must be initialised independently of the execution mode (interpreted/compiled), before it can be evaluated. This also holds for the lambda function body.

Two data structures are created during the built-in initialisation: an ExprState and an ExprContext. The ExprState contains the opcode sequence needed to evaluate the expression as described in Section 3.6. The opcode sequence is generated from the expression tree bottom-up. During the generation of the opcodes, PostgreSQL allocates memory for the results and stores fixed pointers to the memory location where the result is to be stored.

The ExprContext is for holding context information for each evaluation, such as placeholders for each of the nodes contained in an expression. Two pointers to each of these data structures are stored in the LambdaExpr node. Finally, the LambdaExpr node is cast to a Datum pointer, which can then be passed to the table function call.

5.4 Lambda Function Evaluation

As outlined in Section 4.3, lambda functions support four different evaluation modes, of which two are based on the existing interpreted expression evaluation and two rely on a new high-performance implementation. This subsection explains the implementation of these four modes, as well as their interplay with table functions.

5.4.1 Table Function Execution. The executor calls table functions as part of the query plan once all the function arguments, including lambda expressions, have been processed. Before a table function can be called, the extension module, which is a shared library, must be located and loaded into memory. It is necessary for the table function to be aware of the data types of its arguments and it must, therefore, correctly cast the returned `Datum` values to their respective pointer types. Subquery arguments can be read either from a tuplestore (for `LAMBDATABLE`) or requested directly from a given subquery plan node (for `LAMBDACURSOR`).

The table functions implemented as part of this study use the `SFRM_Materialize` mode for returning tuples, which requires output tuples to be written to a tuplestore. This will avoid overhead, unlike with the `ValuePerCall` mode, where the table function is called repeatedly until no more tuples are produced. Furthermore, the table functions return a tuple descriptor, representing the row format of the returned tuples. This tuple descriptor must match the descriptor provided in the column definition list in the SQL query.

5.4.2 Interpreted Execution. The most basic level of lambda evaluation implemented as part of the PostgreSQL lambda extension is the one on which the opcode sequence is processed without any optimisations apart from the trivial simplifications performed by the planner. Interpreted execution is best suited to very small input sizes in time-critical contexts, where the overhead needed for additional optimisation would considerably increase the response times.

5.4.3 JIT-Compiled Execution. Since version 11, PostgreSQL has been shipped with a JIT compilation interface for evaluating expressions as outlined in Section 3.5. Rather than processing the opcode sequence in each evaluation, the sequence is transformed to LLVM IR bitcode. PostgreSQL offers different levels of optimisation for the generated code prior to its execution. In particular, a major performance enhancement can be achieved through inlining.

The decision, as to which function calls should be inlined is based on a cost model and takes into account call stack depth and function complexity. A fixed threshold balances compilation time and execution time: once the accumulated cost of all functions inlined so far exceeds the threshold, no more functions are inlined. The actual inlining is done utilising LLVM bitcode, which is generated during the compilation of PostgreSQL. The functions not eligible for inlining are transformed into plain C function calls.

The final LLVM module consists of a function which evaluates the expression as well as all of the inlined functions it depends on. The module is then compiled using LLVM ORC[4], which provides a modular interface for JIT compilation. An additional optimisation step (O3 level optimisation) may be applied to the generated code. Finally, a function pointer can be obtained from LLVM ORC. The function expects an `ExprState` and an `ExprContext` and will return the result as a `Datum` pointer.

This mode is best suited for algorithms, for which the induced compilation overhead is negligible in comparison to the performance benefit gained for large numbers of expression evaluations.

5.4.4 High-Performance JIT-Compiled Execution. The JIT-compilation framework shipped with PostgreSQL has two major drawbacks: first, the compiled expressions lack thread safety. This is due to the way the opcode sequence is generated from the expression trees: every node in an expression tree will first generate the code needed to compute the results for its child nodes, for which fixed memory positions are allocated. The opcode entries are C structures with a `Datum*` pointer to the memory position at which the result will be stored during execution.

The second drawback is due to the complexity of the generated code. Each PostgreSQL function or operator is backed by a C function implementing the operation. The built-in implementations perform various type checks and validations, including multiple sub-calls, prior to actually computing the desired result.

To overcome these issues, a new JIT framework based on LLVM specifically for lambda expressions was introduced as part of this study. The new system supports thread-safe lambda expressions, which implement a small subset of all the PostgreSQL expression opcodes covering most use cases for lambda expressions. Using only primitive LLVM data types and mathematical function calls, a near hard-coded level of performance can be achieved. In a similar way to the built-in JIT compilation, the new system compiles an evaluation function, which returns the result in row-major order.

The expression tree generates the opcode entries of the lambda function in postfix order, to allow a stack-based buildup of the LLVM structure. For each step in the sequence, a given opcode type will take a certain number of elements from the stack, process them (i.e. wrap them in other LLVM structures) and place zero or more new elements on the stack. The new code generator provides an implementation only for the opcode types needed for our lambda expressions (namely, field selections, constants and function calls). These types, along with the number of their input and output stack elements, are shown in Table 1.

Opcode Type	# in	# out	Description
EEOP_FIELDSELECT[i]	1	1	Takes the row value from the top of the stack and places the i-th field from it on the stack
EEOP_PARAM_EXTERN[i]	0	1	Places the i-th lambda function argument as a row value on the stack.
EEOP_CONST[c]	0	1	Places a constant c on the stack.
EEOP_FUNCEXPR[o]	n	1	Takes n values from the stack, calls the function with Oid o with the values as arguments and places the result on the stack.
EEOP_DONE	1	0	Returns the value on top of the stack.

Table 1: Opcode subset supported by the high-performance JIT compiler.

Once these functions have been implemented, the generated LLVM structure can be compiled and optimised using the LLVM ORC. Figure 7 shows the optimised LLVM IR generated from a lambda expression calculating an L2 distance metric $(a.x - b.x)^2 -$

[4] https://llvm.org/docs/ORCv2.html

$(a.y - b.y)^2$ between two points. Note that the calls to `pow()` have been respectively replaced by two `fmul()` instructions.

```
LAMBDA(a,b)((a.x - b.x)^2 + (a.y - b.y)^2)
```

Original code:

```
define i64 @evalexpr_4_0(i64**) #0 {
entry:
 %1=getelementptr i64*,i64** %0,i32 0
 %2=load i64*,i64** %1
 %3=getelementptr i64,i64* %2,i32 0
 %4=load i64,i64* %3
 %5=getelementptr i64*,i64** %0,i32 1
 %6=load i64*,i64** %5
 %7=getelementptr i64,i64* %6,i32 0
 %8=load i64,i64* %7
 %9=bitcast i64 %8 to double
 %10=bitcast i64 %4 to double
 %fsub=fsub double %10,%9
 %11=call double @pow(double %fsub,
        double 2.000000e+00)
 %12=getelementptr i64*,i64** %0,i32 0
 %13=load i64*,i64** %12
 %14=getelementptr i64,i64* %13,i32 1
 %15=load i64,i64* %14
 %16=getelementptr i64*,i64** %0,i32 1
 %17=load i64*,i64** %16
 %18=getelementptr i64,i64* %17,i32 1
 %19=load i64,i64* %18
 %20=bitcast i64 %19 to double
 %21=bitcast i64 %15 to double
 %fsub1=fsub double %21,%20
 %22=call double @pow(double %fsub1,
        double 2.000000e+00)
 %fadd=fadd double %11,%22
 %23=bitcast double %fadd to i64
 ret i64 %23
}
declare double @pow(double,double)
```

O3 optimized code:

```
define i64 @evalexpr_4_0(i64**
       nocapture readonly)
       local_unnamed_addr #0 {


entry:


 %1=bitcast i64** %0 to double**
 %2=load double*,double** %1,align 8
 %3=load double,double* %2,align 8
 %4=getelementptr i64*,i64** %0,i64 1
 %5=bitcast i64** %4 to double**
 %6=load double*,double** %5,align 8
 %7=load double,double* %6,align 8
 %fsub=fsub double %3,%7
 %square=fmul double %fsub,%fsub
 %8=getelementptr double,double* %2,
       i64 1
 %9=load double,double* %8,align 8
 %10=getelementptr double,double* %6,
       i64 1
 %11=load double,double* %10,align 8
 %fsub1=fsub double %9,%11
 %square2=fmul double %fsub1,%fsub1
 %fadd=fadd double %square,%square2
 %12=bitcast double %fadd to i64
 ret i64 %12


}

declare double @pow(double,double)
```

Figure 7: LLVM IR code generated from a lambda function calculating the L2 distance between two points.

5.4.5 High-Performance JIT Injection (L4). We improved the code generator introduced in the previous subsection by introducing a JIT-injection API (see Figure 8). Directly injecting the code of a lambda expression into one of the table functions avoids external function calls. The LLVM IR generated from the C source code is modified to allow JIT injection.

The PostgreSQL build system automatically generates the corresponding bitcode files (`.bc`) from the table function files. At runtime, the bitcode of the desired table function (its C function name is specified as a string) is first loaded as an LLVM module. It is assumed that all lambda functions passed to the table function have already been built and stored in another LLVM module. To actually inject the lambda expression, an LLVM function pass needs to recognise the locations at which the lambda evaluation is to be executed.

First, the modules of the lambda function and the table function are merged into a common one. Once all IR instructions have been checked, calls to the lambda evaluation functions replace the function call instructions stored in a temporary list. Finally, a function pointer to the table function is retrieved and can be called right away.

5.4.6 Row-Type Deduction. To avoid specifying a column definition list explicitly, table functions should be able to deduce subquery types automatically. Internally, this is solved during the semantic analysis using a helper function. This helper function receives a tuple descriptor for the two new types, `LAMBDATABLE` and `LAMBDACURSOR` and returns another one as input for the table function.

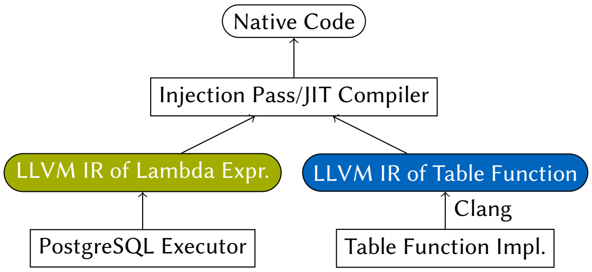


Figure 8: Structure of the JIT injection API.

6 TABLE FUNCTION IMPLEMENTATIONS

In this study, three algorithms were implemented as table functions to demonstrate and evaluate PostgreSQL with lambda expressions and enhanced JIT compilation. These are labelling, k-Means clustering and PageRank computation. This section outlines the implementation details for these table functions.

6.1 Labelling

The labelling function adds a label given as a lambda expression to existing data. It takes one input relation and one lambda expression as arguments and returns all input tuples with the result of the lambda function appended to the tuple as an additional column. For this problem, we choose the type `LAMBDACURSOR` as a subquery, since the algorithm only needs to iterate over the input tuples once and we do not need to initialise a fixed-size data structures before execution.

```
postgres=# SELECT * FROM label_fast((SELECT x, y FROM points LIMIT
       10), LAMBDA(a)(sqrt(a.x^2 + a.y^2)));
       x         |         y         |      label
-470.860150642693|-124.311549589038|486.993473082019
-467.270653229207|-432.609483599663|636.783188117915
```

Figure 9: Example output of the labelling function.

Two variants are implemented as labelling functions: `label` and `label_fast`. The former uses basic JIT-compiled execution, while the latter uses high-performance JIT-compiled execution and is, therefore, faster, but limited to numerical data types. Figure 9 shows an example output of a `label_fast` call. In compliance with the API defined in Section 5.4.6, both functions offer a common row-type deduction function, which simply appends a column named `label` to the tuple descriptor of the input record type. The return type of the lambda function determines the data type of the added column.

6.2 PageRank

PageRank is a graph mining algorithm that labels nodes according to their incoming and outgoing vertices. Page and Brin [19] developed the algorithm as the foundation of Google's search engine for ranking websites. Each node receives a PageRank value, that devolves to connected nodes in each iteration. After convergence, the final PageRank value corresponds to the node's importance.

The PostgreSQL table function for computing PageRank values expects the following parameters:

- The subquery yielding the input tuples T,
- two lambda functions λ_{src} and λ_{dst} selecting the source and destination node identifier from an input tuple,
- the damping factor α and
- a threshold t for termination of the algorithm.

T describes the input tuples that store information about the vertices. Each lambda expression, λ_{src} and λ_{dst}, selects the input and output node per tuple that can be constructed using arbitrary expressions. They form a set of edges (Equation 3) representing the links-to relationship between two given nodes. Implicitly, they also define the set of nodes (Equation 4).

Initially, the PageRank for each node is equally distributed (Equation 5) so that the sum of all values is one. Afterwards, the PageRank values are iteratively computed for all nodes (Equation 6). The new PageRank is the sum of the proportionate PageRank values of all nodes connected via incoming edges, with α as the damping factor:

$$E = \{(\lambda_{src}(e_1), \lambda_{dst}(e_2)) \mid e_1, e_2 \in T\}, \tag{3}$$

$$P := \{s \mid \exists d : (s, d) \in E\} \cup \{d \mid \exists s : (s, d) \in E\}, \tag{4}$$

$$PR_0(d) := \frac{1}{|P|}, \tag{5}$$

$$PR_{i+1}(d) := \alpha \cdot \sum_{(s,d) \in E} \frac{PR(s)_i}{|\{p \mid (s, p) \in E\}|} + \frac{1 - \alpha}{|P|}. \tag{6}$$

The input tuples T are passed to the table function as a LAMBDATABLE parameter. This allows the function to correctly allocate the amount of memory to the data structures that is needed for the algorithm before the input data is read. To enable efficient computation, the edges are stored in a compressed sparse row format similar to HyPer [20]. This also involves mapping the nodes to dense integer values (see Figure 10). The resulting sparse row encoding of the example is shown in Figure 11. The row is a linear array of node identifiers. Each node is assigned a starting position in the row from which the identifiers of its incoming nodes can be read.

src	dst
30786325628624	4194
32985348837431	4194
32985348867163	4194
32985348878771	4194
22854	8333
55093	8333
2199023273826	8333

→

Original key	New key
30786325628624	0
4194	1
32985348837431	2
32985348867163	3
32985348878771	4
22854	5
8333	6
55093	7
2199023273826	8

Figure 10: Edge table and dictionary for dense relabelling.

The relabelling step is implemented using the built-in PostgreSQL hash table data structure as a dictionary. The dictionary has as a key the node identifier returned by the two lambda functions and returns a pointer to a C struct holding all the important metadata such as the original key and the number of outgoing edges.

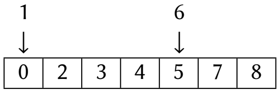


Figure 11: Example of compressed sparse row encoding: the node with the new key 1 **is connected via incoming edges to the nodes** 0, 2, 3, 4**.**

Multiple worker threads compute the actual PageRank. Each of the n worker threads is assigned (at most) $\lceil \frac{|P|}{n} \rceil$ nodes, by which to compute the next PageRank value according to the update rule defined above. The new PageRank value is stored inside the C struct of the corresponding page. To avoid concurrent read or write accesses, each struct has two values for its PageRank, which alternatingly take the roles of $PR(p)_i$ and $PR(p)_{i+1}$. After each iteration, the main thread waits until all workers have finished their computations and writes the new PageRank values out. The computation ends when the change for every node falls below a certain threshold: $\max_{p \in P} |PR(p)_i - PR(p)_{i+1}| \leq t$.

Finally, the original keys are restored and the result set is populated. The returned row type consists of one column named node for the nodes and a float8 column named PageRank, which holds the final PageRank values. Figure 12 shows an example output of the PageRank function for a person-knows-person data set.

```
postgres=# SELECT * FROM pagerank((SELECT src,dst FROM knows),
      LAMBDA(src)(src.src), LAMBDA(dst)(dst.dst), 0.9, 0.001, 45)
      ORDER BY pagerank DESC;
     node       |       pagerank
28587302384882 |0.000403720299288661
 4398046574506 | 0.00039321123228248
26388279120130 |0.000390113303915129
```

Figure 12: Example output of the PageRank function.

6.3 k-Means Clustering

The third table function implemented as part of this study is the k-Means clustering algorithm as described by Lloyd [16]. Given a set of k 2-dimensional points $P = \{p_1, ..., p_k\}$ and m initial cluster centres $C = \{c_1, ..., c_m\} \subset P$, each point is assigned to its closest cluster centre. Afterwards, the cluster centres are set to the centre of all the points assigned to it. This process is repeated until the cluster assignments converge. The corresponding table function accepts the following arguments:

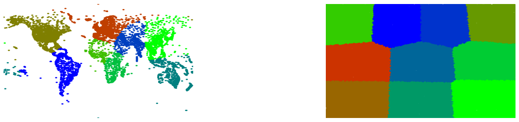
- The subquery yielding the input points P,
- the subquery yielding the initial cluster centres C,
- a lambda function λ_{dist}, which calculates the distance between two given points, and
- the expected number m of clusters.

Both input subqueries need to have the same row format and are loaded as LAMBDATABLE arguments. This implementation supports two-dimensional points and requires the x and y coordinates to be given as float8 values. The lambda function λ_{dist} must, therefore, accept two tuples and return the result as a float8 value.

Initially, the points and cluster centres are loaded into pre-allocated arrays, which comply with the Datum** parameter format

```
postgres=# SELECT * FROM kmeans((SELECT (latitude / 180 * pi()) AS
      lat, (longitude / 180 * pi()) AS lng, rowid FROM airports
     LIMIT 8), (SELECT (latitude / 180 * pi()) AS lat, (longitude
     / 180 * pi()) AS lng, rowid from airports limit 1500000),
     LAMBDA(a,b)(2.0 * atan2(sqrt(sin((b.lat-a.lat)/2.0) ^ 2.0 +
     cos(a.lat) * cos(b.lat) * (sin((b.lng - a.lng) / 2.0) ^ 2.0))
     , sqrt(1-sin((b.lat-a.lat)/2.0) ^ 2.0 + cos(a.lat) * cos(b.
     lat) * (sin((b.lng - a.lng) / 2.0) ^ 2.0)))), 8);
      lat          |        lng         |rowid|cluster
-0.0908806884493311|  2.54449808875909|   2 |      2
 -0.101696667808256|  2.51844038907048|   3 |      2
 -0.114664693557401|  2.56085139685547|   4 |      2
 -0.164818079727474|  2.56947374609134|   5 |      2
-0.0625496353943785|  2.50749719384123|   6 |      2
   1.06745208970995|-0.792833243106988|   7 |      7
```

Figure 13: Textual k-Means output for an airport data set.



(a) Airport data set. **(b) Uniformly distributed points.**

Figure 14: Visualised k-Means output.

expected by the high-performance lambda expressions. The average for the cluster centres is stored separately by each worker thread and only written back after each iteration, which allows concurrent read access to the coordinates.

Just like the PageRank table function, the k-Means function supports multi-threaded computation and spawns n worker threads. However, for this task, the distance function λ_{dist} is injected directly into the worker thread code, using the thread-safety of the JIT injection framework explained earlier. Each of the n workers processes (at most) $\lceil \frac{|P|}{n} \rceil$ points in each iteration. The point-cluster distance is computed for each point/cluster pair, and the point is then assigned to the cluster with the minimum distance:

$$c_{i+1}(p) := \arg\min_{c \in C} \lambda_{dist}(p, c). \qquad (7)$$

The distance function specified as a lambda expression allows various use cases and does not limit the algorithm to a specific type of coordinate system: A Euclidean distance function may be specified for (x, y) coordinate pairs, whereas a haversine formula might be provided to compute a geographical distance between two latitude/longitude coordinates. The row format returned by the table function consists of all input columns of point p with one additional integer column specifying the index $c(p)$ of the cluster the point has been assigned to.

Figure 13 shows the SQL query for the k-Means table function applied to an airport data set using the haversine formula as a lambda function. Figure 14 shows the visualisation of clustering with k-Means for the airport data set as well as for uniformly distributed Euclidean points, effectively producing a Voronoi diagram.

7 EVALUATION

This section discusses the evaluation of the PostgreSQL lambda extension. For evaluation, we benchmarked the table functions with the generated data as well as real-world data sets, varying the number of input tuples or the available threads. We ran k-Means with ten clusters and terminated after 80 iterations. For PageRank, the damping factor α was set to 0.85 and the threshold to 0.00001 and we terminated the computation after a minimum of 100 iterations.

7.1 Data Sets

The LDBC Social Network Benchmark[5] data set was used for the PageRank evaluation. It simulates activity on a social network and includes a person-knows-person relation. The data set used for this evaluation was generated by the LDBC data generator script with a scale factor of 10, resulting in $1.9 \cdot 10^6$ edges for the person-knows-person relation. The PageRank algorithm without damping was used to determine the "best-known" persons in the database.

An excerpt from the Chicago taxi trip data set[6] was used to evaluate the k-Means clustering. It contains all Chicago taxi rides, including drop-off locations, given as latitude and longitude coordinates. An additional data set for the k-Means clustering algorithm consisting of $2 \cdot 10^7$ uniformly distributed Euclidean points in $[-500.0, +500.0]$ was generated directly from an SQL script. These points were also used for evaluating the labelling function.

We also tested the labelling function with generated data (using the built-in `generated_series` function) of various sizes.

7.2 Test Environment

All experiments were run on a Ubuntu 18.04.3 LTS machine with an Intel Xeon E5-2660 v2 (2.20GHz) processor with 20 cores/40 threads and approximately 252 GiB of main memory. The (modified) PostgreSQL database (version 11.2) was compiled on the machine with LLVM 7 for JIT support. In addition, a HyPer instance supporting the new lambda operators [24] was running on the server for comparative measurements.

Each test was executed five times and the results were averaged. To render the results more comparable to HyPer, which is purely a main-memory database, the `work_mem` configuration parameter of PostgreSQL was set to 8 GB. This prevents PostgreSQL from writing out any tuplestores to disk, by keeping it working only in main memory.

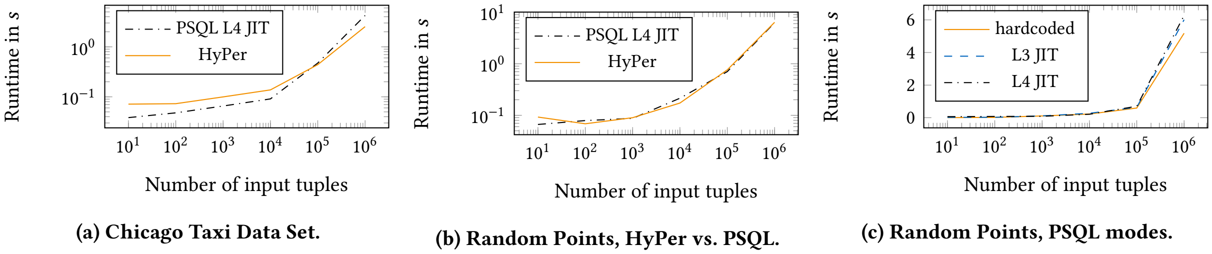
Most of the measurements were conducted from a Python script, using the psycopg2 extension for database access. For both HyPer and PostgreSQL, a local Unix socket was used for the connection. The measurement time spanned the entire execution time including parsing and compilation time. To measure the JIT compilation time and actual execution time separately in PostgreSQL, the compilation times were measured internally and written to a log file.

7.3 Varying the Input Size

The first tests evaluated the three table functions (labelling, k-Means clustering and PageRank) with different input sizes. A comparative measurement was run on the equivalent query in the HyPer database. The input sizes were varied by specifying a respective `LIMIT` for the subqueries when loading the input data.

[5] https://github.com/ldbc/ldbc_snb_datagen

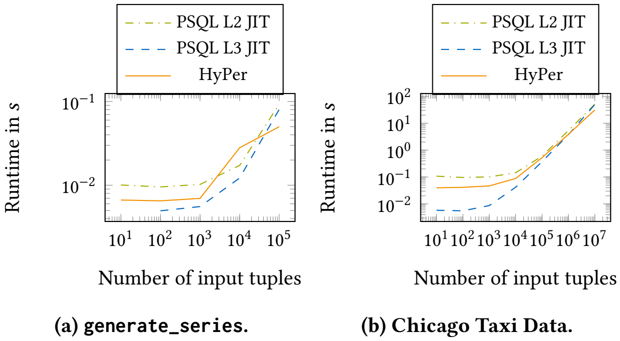
[6] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew



(a) Chicago Taxi Data Set. **(b) Random Points, HyPer vs. PSQL.** **(c) Random Points, PSQL modes.**

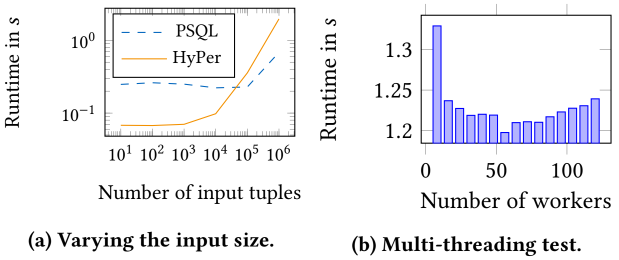
Figure 15: Varying the input size for k-Means where $k = 10$, with two dimensions and 80 iterations: (a) with the Chicago taxi data set and (b) with randomly generated points; (c) compares the different compilation modes.

Both high-performance (L3) and basic JIT-compiled (L2) lambda expressions were tested for the label function (see Figure 16). The high-performance implementations were highly competitive with respect to their HyPer counterparts, whereas HyPer outperformed the built-in PostgreSQL JIT compilation for larger inputs.



(a) `generate_series`. **(b) Chicago Taxi Data.**

Figure 16: Varying the input size for labelling.



(a) Varying the input size. **(b) Multi-threading test.**

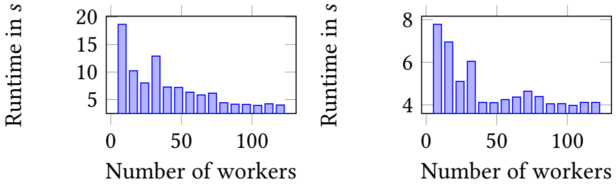
Figure 17: PageRank: LDBC Person-Knows-Person data set, 10^6 tuples, L2 JIT, $\alpha = 0.85$, 100 iterations.

For k-Means, we evaluated L3 and L4 evaluation modes (see Figure 15). In a separate experiment, the PostgreSQL lambda evaluation was compared with a hard-coded implementation, which was optimised with O3 level by the compiler. For this purpose, the computations done by the lambda expression were hard-coded directly in the table function. Figure 15c shows that the overhead induced by the JIT-compilation was very low and barely measurable for smaller inputs.

When varying the number of input tuples, the evaluation for PageRank (see Figure 17a) with the LDBC data set reveals a constant overhead of about 250 ms, which is caused during preprocessing when creating a dictionary. This problem could be solved by optimising the preprocessing with modern data structures that support multi-threaded reads and writes rather than the built-in PostgreSQL hash table implementation. For the PageRank test, the PostgreSQL query had an additional `ORDER BY PageRank DESC` clause attached, accounting for the fact that the HyPer implementation returns the tuples sorted, which the PostgreSQL implementation does not do.

7.4 Multi-Threading Tests

For the multi-threaded implementations of the k-Means and PageRank functions, modified versions of these functions were implemented. These allow the number of worker threads to be varied directly from SQL. Based on the test results shown in Figure 18, it can be observed that for the uniformly distributed k-Means test, run time decreases by approximately 20 % with every eight additional threads, until saturation occurs from 80-90 threads onwards. This conforms with the number of logical cores of the evaluation server and shows that the algorithms make efficient use of all available cores. For the Chicago taxi trips data set, saturation already occurs from 40 threads onwards due to the overall cluster convergence being considerably faster than for uniformly distributed data.



(a) Randomly generated points. **(b) Chicago taxi data set.**

Figure 18: Multi-threading tests for k-Means where $k = 10$, with two dim., 80 iterations and (a) $2 \cdot 10^7$ or (b) 10^6 tuples.

The performance benefits achieved through multi-threading were not as high for PageRank as for k-Means, as illustrated by Figure 17b. This is mainly due to the fact that the most time-consuming tasks in PageRank are the preprocessing/relabelling steps, which are not multi-threaded in this implementation. The actual PageRank algorithm iterations take up only a very small percentage of the total execution time.

8 CONCLUSION

This study has successfully introduced a way of integrating lambda expressions in the open-source database system PostgreSQL. First, the possibilities of using lambda expressions in database systems were illustrated based on the existing HyPer system with its high-performance data mining operators. In order to integrate lambda functions in PostgreSQL, its grammar had to be adjusted to accept the new lambda expression syntax. The PostgreSQL extension system made for a good foundation for the new table functions in supporting lambda expressions, as they allowed functions to be implemented in C and called from SQL. However, one major obstacle emerged from the way subqueries were handled internally: subqueries were not able to return more than one column when passed as a parameter to a table function. This problem was successfully solved by introducing two new types of subqueries, which could be read from table functions. Furthermore, table functions using one of the new subquery types were equipped with automatic row-type deduction, rendering column definition lists in SQL unnecessary, thus making the usage of the new table functions more convenient.

PostgreSQL already supported basic JIT compilation, so it was taken as the starting point for the new lambda function evaluation. This study introduced four modes for evaluating lambda expressions, two of them using the existing JIT compilation system and the other two using a new JIT compilation framework backed by lightweight LLVM expressions. This idea was partly inspired by the query compilation method found in the HyPer database system, in which queries are compiled entirely into native code before they are executed. Three exemplary data mining algorithms, namely, k-Means, labelling and PageRank, were implemented as table functions, using the new lambda expression system. Evaluation with different types of test data sets revealed highly competitive levels of performance and scalability. The extension can therefore be regarded as a promising foundation for a variety of data mining tasks in the PostgreSQL database system.

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). arXiv:1603.04467 http://arxiv.org/abs/1603.04467

[2] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2017. Mind the Gap: Bridging Multi-Domain Query Workloads with EmptyHeaded. *PVLDB* 10, 12 (2017), 1849–1852. http://www.vldb.org/pvldb/vol10/p1849-aberger.pdf

[3] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. 2012. Theano: new features and speed improvements. *CoRR* abs/1211.5590 (2012). arXiv:1211.5590 http://arxiv.org/abs/1211.5590

[4] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. 2015. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper18.pdf

[5] Mike Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Mueller, Sonia Castelo, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it. In *CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p13-brachmann-cidr20.pdf

[6] Dennis Butterstein and Torsten Grust. 2016. Precision Performance Surgery for PostgreSQL: LLVM-based Expression Compilation, Just in Time. *PVLDB* 9, 13 (2016), 1517–1520. https://doi.org/10.14778/3007263.3007298

[7] Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 2 (April 1936), 345–363. https://doi.org/10.2307/2371045

[8] Bin Dong, Patrick Kilian, Xiaocan Li, Fan Guo, Suren Byna, and Kesheng Wu. 2019. Terabyte-scale Particle Data Analysis: An ArrayUDF Case Study. In *SSDBM 2019, Santa Cruz, CA, USA, July 23-25, 2019.* ACM, 202–205. https://doi.org/10.1145/3335783.3335805

[9] Christian Duta, Denis Hirn, and Torsten Grust. 2020. Compiling PL/SQL Away. In *CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p1-duta-cidr20.pdf

[10] Ahmed Eldawy, Ibrahim Sabek, Mostafa Elganainy, Ammar Bakeer, Ahmed Abdelmotaleb, and Mohamed F. Mokbel. 2017. Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data. In *SSTD 2017, Arlington, VA, USA, August 21-23, 2017, Proceedings (Lecture Notes in Computer Science)*, Vol. 10411. Springer, 65–83. https://doi.org/10.1007/978-3-319-64367-0_4

[11] Maxim Filatov and Verena Kantere. 2016. PAW: A Platform for Analytics Workflows. In *EDBT 2016, Bordeaux, France, March 15-16, 2016.* OpenProceedings.org, 624–627. https://doi.org/10.5441/002/edbt.2016.64

[12] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135. https://doi.org/10.1109/69.273032

[13] Ali Hadian and Thomas Heinis. 2019. Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes. In *EDBT 2019, Lisbon, Portugal, March 26-29, 2019.* OpenProceedings.org, 710–713. https://doi.org/10.5441/002/edbt.2019.93

[14] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB* 5, 12 (2012), 1700–1711. http://vldb.org/pvldb/vol5/p1700_joehellerstein_vldb2012.pdf

[15] Nina Hubig, Linnea Passing, Maximilian E. Schüle, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. 2017. HyPerInsight: Data Exploration Deep Inside HyPer. In *CIKM 2017, Singapore, November 06 - 10, 2017.* 2467–2470. https://doi.org/10.1145/3132847.3133167

[16] Stuart P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Information Theory* 28, 2 (1982), 129–136. https://doi.org/10.1109/TIT.1982.1056489

[17] Dmitry Melnik, Ruben Buchatskiy, Roman Zhuykov, and Eugene Sharygin. 2017. JIT-compiling SQL queries in PostgreSQL using LLVM.

[18] Dimitar Misev and Peter Baumann. 2014. Extending the SQL array concept to support scientific analytics. In *SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014.* ACM, 10:1–10:11. https://doi.org/10.1145/2618243.2618255

[19] L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. The PageRank citation ranking: Bringing order to the Web. In *WWW*. Brisbane, Australia, 161–172. citeseer.nj.nec.com/page98pagerank.html

[20] Linnea Passing, Manuel Then, Nina Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *EDBT 2017, Venice, Italy, March 21-24, 2017.* OpenProceedings.org, 84–95. https://doi.org/10.5441/002/edbt.2017.09

[21] Daniel Popovic, Edouard Fouché, and Klemens Böhm. 2019. Unsupervised Artificial Neural Networks for Outlier Detection in High-Dimensional Data. In *ADBIS 2019, Bled, Slovenia, September 8-11, 2019, Proceedings (Lecture Notes in Computer Science)*, Vol. 11695. Springer, 3–19. https://doi.org/10.1007/978-3-030-28730-6_1

[22] Maximilian Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *EDBT 2019, Lisbon, Portugal, March 26-29, 2019.* OpenProceedings.org, 562–565. https://doi.org/10.5441/002/edbt.2019.56

[23] Maximilian Schüle, Linnea Passing, Alfons Kemper, and Thomas Neumann. 2019. Ja-(zu-)SQL: Evaluation einer SQL-Skriptsprache für Hauptspeicherdatenbanksysteme. In *BTW 2019, 4.-8. März 2019, Rostock, Germany, Proceedings.* 107–126. https://doi.org/10.18420/btw2019-08

[24] Maximilian Schüle, Dimitri Vorona, Linnea Passing, Harald Lang, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. The Power of SQL Lambda Functions. In *EDBT 2019, Lisbon, Portugal, March 26-29, 2019.* OpenProceedings.org, 534–537. https://doi.org/10.5441/002/edbt.2019.49

[25] Jun Hyung Shin, Florin Rusu, and Alex Suhan. 2019. Selectivity Computation for In-Memory Query Optimization. In *CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2019/gongshow/abstracts/cidr2019_82.pdf

[26] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD Conference, Washington, DC, USA, May 28-30, 1986.* 340–355. https://doi.org/10.1145/16894.16888

[27] Thanh Truong and Tore Risch. 2015. Transparent inclusion, utilization, and validation of main memory domain indexes. In *SSDBM '15, La Jolla, CA, USA, June 29 - July 1, 2015.* ACM, 21:1–21:12. https://doi.org/10.1145/2791347.2791375

[28] Haoyuan Xing, Sofoklis Floratos, Spyros Blanas, Suren Byna, Prabhat, Kesheng Wu, and Paul Brown. 2018. ArrayBridge: Interweaving Declarative Array Processing in SciDB with Imperative HDF5-Based Programs. In *ICDE 2018, Paris, France, April 16-19, 2018.* IEEE Computer Society, 977–988. https://doi.org/10.1109/ICDE.2018.00092