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 to inject user-defined metrics into data mining operators during compile time. Since version 11, PostgreSQL supports just-in-time compilation with LLVM for expression evaluation. This allows transferring the concept of SQL lambda functions to this open-source database system.

In this work, we extend PostgreSQL with two additional sub-query 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 function for data mining algorithms such as PageRank, k-Means clustering and labelling. Furthermore, we allow four levels of optimisation for the query execution, that range from interpreted function calls to just-in-time-compiled execution. The latter—with some adjustments in the PostgreSQL’s execution engine—turns 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 achieve comparable performance to hard-coded implementations and HyPer’s data mining algorithms.

1 INTRODUCTION
Shifting computations to database systems is one of the main-challenges that research on database systems is focussed on [4, 11]. To shift the boundary between database systems and dedicated tools [5], extensions such as MADlib [14] allow in-database analytics with dedicated table operators inside of relational database systems but 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 pre-compiled and injected into otherwise inflexible operators (see Figure 1). This is possible as HyPer is generating code using the LLVM (low-level virtual machine) compiler backend. So far, as the research was focused on HyPer, the concept of lambda functions has been published for the specific case of a modern main-memory database system only and the corresponding source code is restricted.

To overcome these limitations, this work shows the adaption of SQL lambda functions to a traditional disk-based database system. In PostgreSQL, we extend its grammar to allow lambda expressions and subqueries as table function’s arguments. Table functions request tuples from subqueries or pre-materialise their results. Similarly to code-generation when evaluating expressions, we seamlessly inject lambda expressions into the table function’s code. In detail, this work’s contributions are:

• an extension of PostgreSQL’s semantic analysis to support lambda functions as table function arguments,
• two different subquery types, as a materialised table or as an operator plan, that can be used inside of table functions,
• 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,
• and an evaluation in terms of scalability when varying the input size or the number of available threads.
This work consists of the following parts: Section 2 describes foundations about lambda expressions in general, work related to lambda functions in Hyper and just-in-time compilation for PostgreSQL. Subsequently, Section 3 explains the PostgreSQL database architecture with special focus 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, especially with regards to optimisation. Section 6 discusses the implementation of the three data mining algorithms, which will make use of the proposed lambda expressions. The evaluation (Section 7) shows the performance of the three data mining algorithms implemented as part of this work with different types of input data. Afterwards, the evaluation results will be compared with an equivalent Hyper test run.

## 2 RELATED WORK

This chapter introduces just-in-time compilation for database systems and tools used for data analysis. They build the foundation for lambda functions in the code-generating database system Hyper, whose table function’s architecture will be adapted for PostgreSQL.

### 2.1 Code-Generation within Database Systems

PostgreSQL [24] is a disk-based relational database system with Volcano-style query execution [12], where the top-most operator demands the underlying ones to produce tuples. Hyper [15] introduced code-generation together with a bottom-up query execution model that pushes tuples upwards the parent operator. Code-generation avoids the overhead of interpreted functions calls by pre-compiling the query first into 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 proposed an own evaluation library that leaves the rest of the PostgreSQL system unchanged. Afterwards, Melnik et al. [17] demonstrated how the internal PostgreSQL functions can be compiled with LLVM to improve execution speed by up to 20%. Finally, in 2018, Andres Freund has released LLVM support for expression evaluation as part of PostgreSQL version 111.

### 2.2 Data Processing Tools

Knowledge discovery on databases, usually done with dedicated tools such as Pytorch, Theano [3] or TensorFlow [1], involves data wrangling, data preprocessing, data analysis or even training a model [13] to deploy neural networks [21]. Whereas the latter one mostly relies on matrix algebra [18, 26], SQL with user-defined functions (UDFs) is well suited especially for data preprocessing and data analysis [10]. 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 such as MADlib [14] for PostgreSQL or as part of the main-memory database systems [23, 25] EmptyHeaded [2] and Hyper [15] achieve better performance for predefined data mining operators. We implement similar table functions for exemplary usage together 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 by Alonzo Church in 1936 [7]. As part of Hyper’s extended SQL, lambda expressions allow users to inject customised code in otherwise inflexible table functions [20, 22]. The basic structure of a SQL lambda function in the Hyper database is constructed as follows:

\[
\lambda(\text{name}_1, \text{name}_2, \ldots)(\text{expr}) .
\]

For the remainder of this work, the term lambda arguments refers to the named arguments between the first pair of parentheses and lambda expression body 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 constructing regular SQL expressions 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:

\[
\lambda(S, T)((S.x - T.x)^2 + (S.y - T.y)^2). 
\]

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 done based on the input row types to the function:

```sql
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
\lambda(a, b) (a.x-b.x)^2+(a.y-b.y)^2 < 2, 3);
```

## 3 THE POSTGRESQL DATABASE SYSTEM

This section explains the components of the database system PostgreSQL, which are modified in order to integrate lambda functions as subqueries and allow compiled execution later on.

### 3.1 Stages of Query Execution

The PostgreSQL query execution involves multiple stages. The raw query as string is passed to the parser that splits the query into tokens according to the PostgreSQL grammar. The analyser then proceeds with the semantic analysis, which consists of building expressions, resolving table names, attributes and function calls and type checking. The modified query tree is then passed to the planner/optimizer that will output an optimised query tree. The plan tree is finally handed over to the executor, which performs the actual table scans, 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 standard. These functions return any built-in type as well as entire tables, then called table functions. For table functions two composite types are possible:

- `TABLE` includes a fixed row type definition,

---


• SETOF RECORD does not include a row type definition in the function definition and is suitable for function 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:

\[
\text{SELECT \ast FROM foo(1, 2) as (a \text{ int}, b \text{ int});}
\]

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

\[
\text{CREATE FUNCTION foo(int, int) RETURNS TABLE (a \text{ int}, b \text{ int}) AS 'foo .so', 'foo' LANGUAGE 'C';}
\]

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

\[
\text{SELECT \ast FROM foo(1, 2);}
\]

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 providing tuple descriptors in the following sections.

### 3.3 Important Data Structures
To follow the implementation details in this work, knowledge of the following PostgreSQL data structures is necessary:

- **Tupletores** are an internal data structure 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 has to 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 is not supposed to be called again.

### 3.5 JIT Compilation
Just-in-time (JIT) compilation with LLVM allows to dynamically generate and compile code on-the-fly from a running program. Since version 11, PostgreSQL supports 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 (\(--\text{with-llvm}\)).

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_EXP** enables 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_DEF** 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 of reimplementing 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 get evaluated in the executor stage. Each expression tree first gets compiled to a sequence of opcodes, that represents elementary operations 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 for interpreted and compiled evaluation.

\[
\text{SELECT } a.x + 2 * a.y \text{ FROM ...}
\]

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

Interpreted evaluation simply loops over all steps and directly performs the computations. Compiled evaluation is utilising 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 the PostgreSQL version 11.2. This subsection describes the high-level concept towards integrating lambda functions in PostgreSQL that consists of the lambda syntax, its evaluation and the integration as parameters in table functions.

4.1 Lambda Function Definitions
The proposed 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 making lambda expressions a dedicated language feature of PostgreSQL. As a dedicated feature, we seamlessly integrate lambda expressions into the existing grammar and type inference system. This will also make the general PostgreSQL expression optimiser aware of the lambda expressions, especially concerning analysing and simplifying the lambda expression body.

Not only the expression body but also the entire lambda function is treated as an expression, as it must be passed as a table function parameter. We introduce a new pseudotype called LAMBDA 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 similarly to MADlib requires another database connection to retrieve the data. However, this is infeasible, since the semantic analysis of the lambda expressions cannot deduce the return type of lambda functions during compile time.

Therefore, we adjust the PostgreSQL subquery system to support multi-column and multi-row subqueries. We 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 explains the difference between the two types.

![Comparison of LAMBDATABLE and LAMBDACURSOR modes for passing input data to table functions.](image)

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, calculating 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 do not know the number of returned tuples beforehand.

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 pre-compile these expressions. This work 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 the 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 get 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 are 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 strongly improve the run time due to a 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 will be 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 as it receives as input, plus one extra column, an integer describing the assigned cluster number. When PostgreSQL cannot deduced 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, that is provided online. The main focus lies on the necessary extensions in the different stages of the PostgreSQL query execution.

5.1 Parser and Analyser Extensions
The extension for lambda expression affects the parser as well as the analyser. We first describe the extension of the PostgreSQL grammar to parse lambda expressions and 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 lambda expressions is formed by a syntax extension of the PostgreSQL grammar. As a first step, the PostgreSQL parser is extended by a new grammar element that is inspired by the HyPer syntax described in Section 2.3 as follows:

\[ \text{LAMBDA}(name_1, name_2, \ldots)(expr). \]

We prefer the LAMBDA keyword to the \( \lambda \) symbol to avoid text encoding conflicts due to a non-ASCII character, 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 can be any regular PostgreSQL expression.

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

Inspecting the syntax defined above, the lambda expression grammar consists of three parts, namely

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

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 might 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, that 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 will be parsed correctly as a type name without any required 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 feasible to represent a lambda expression definition as a Expr node as well since it is considered 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 parsing, analysing and execution stages. For this purpose, a new node named LambdaExpr is introduced. This data structure will not only hold parse information but is also being passed to table functions, allowing them to perform additional type checks.

![Figure 4: Grammar rules for parsing a lambda expression definition in Backus-Naur form.](https://gitlab.db.in.tum.de/JakobHuber/postgres-lambda-diff)

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 for the argument names (stored in the arg field) as well as an Expr node for the expression body is created. 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 case of syntax errors. These three fields are the only ones filled during parsing.

5.1.3 Type Definitions. With the grammar rules for lambda expressions defined, the next step is to register a new lambda type in the system catalogues. Since lambda expressions should become a deeply integrated feature of the PostgreSQL database, the lambda type...
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) with its name together with 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 of 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 LAMBDAABLE and LAMBDACURSOR (as explained in Section 4.2) are registered as pseudo-types in the same fashion.

CREATE OR REPLACE FUNCTION foo (LAMBDAABLE 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.

With these three new types registered, it is now possible to specify LAMBDA and the two additional types LAMBDACURSOR respectively LAMBDAABLE as parameter types 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 being a LAMBDAABLE and the other one being a LAMBDACURSOR) and one lambda expression. The types are generic and do not define any kind of restriction concerning 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. With the syntax extension and type definitions 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 errors or other errors, which cannot be detected by a simple syntax check, and resolve tables, function calls and other variables used in the query. The necessary adjustments for integrating lambda expressions especially 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 or 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 the two new lambda subquery types: the LAMBDAABLE type needs 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 LAMBDAABLE 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 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 when preparing the arguments for the table function call. Each PostgreSQL expression, independent of the execution mode (interpreted/compiled), must be initialised before it can be evaluated. This also holds for the lambda function body.

During the built-in initialisation, two data structures are created: 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 supposed to be stored.

The ExprContext will hold context information for each evaluation, such as placeholders for each node 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 may 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 will call table functions as part of the query plan after 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. The table function is required to be aware of the data types of its arguments and must,
because, cast the returned Datum values to their proper pointer types. Subquery arguments can be read either from a tuplestore (for LAMBDA_TABLE) or requested directly from a given subquery plan node (for LAMBDA_CURSOR).

The table functions implemented as part of this work use the SFRM_Materialize mode for returning tuples, which requires output tuples to be written to a tuplestore. This will avoid overhead as opposed to the VALUE_PER_CALL mode where the table function gets 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 processing the opcode sequence without any optimisations apart from the trivial simplifications performed by the planner. Interpreted execution is best suited for 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 is shipped with a JIT compilation interface for evaluating expressions as outlined in Section 3.3. Rather than processing the opcode sequence in each evaluation, the sequence is transformed to LLVM IR bytecode. PostgreSQL offers different levels of optimising the generated code before executing it. A major performance benefit is achieved through inlining.

The decision, which function calls should be inlined is based on a cost model, accounting for 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 utilizing LLVM bytecode, 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 and all of the inlined functions it depends on. The module is then compiled using LLVM ORC4, that 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 owed 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 the result will be stored during execution.

The second drawback lies within 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 before actually computing the desired result.

To overcome these issues, a new JIT framework based on LLVM specifically for lambda expressions is introduced as part of this work. 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 performance can be achieved. Similarly 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 build-up of the LLVM structure. For every 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 for the opcode types only needed for our lambda expressions (namely, field selections, constants and function calls). These types along with their number of input and output stack elements are shown in Table 1.

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

<table>
<thead>
<tr>
<th>Opcode Type</th>
<th># in</th>
<th># out</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOP_FIELDSELECT[i][j]</td>
<td>1</td>
<td>1</td>
<td>Takes the row value from the top of the stack and places the i-th field from it on the stack.</td>
</tr>
<tr>
<td>EOP_PARAM_EXTERN[i]</td>
<td>0</td>
<td>1</td>
<td>Places the i-th lambda function argument as a row value on the stack.</td>
</tr>
<tr>
<td>EOP_CONST[c]</td>
<td>0</td>
<td>1</td>
<td>Places a constant c on the stack.</td>
</tr>
<tr>
<td>EOP_FUNEXPR[o]</td>
<td>n</td>
<td>1</td>
<td>Takes n values from the stack, calls the function with Oid o with the values as arguments and places the result on the stack.</td>
</tr>
<tr>
<td>EOP_DONE</td>
<td>1</td>
<td>1</td>
<td>Returns the value on top of the stack.</td>
</tr>
</tbody>
</table>

With these implementations, the generated LLVM structure can then be compiled and optimised with the LLVM ORC. Figure 7 shows the optimised LLVM IR generated from a lambda expression calculating a L2 distance metric \((a \cdot x - b \cdot y)^2 - (a \cdot y - b \cdot x)^2\) between two points. Note that the calls to `pow()` have been replaced by two `fma()` instructions respectively.

5.4.5 High-Performance JIT Injection (L4). We improve 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 the one of the table function 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
As part of this work, three algorithms have been implemented. 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.

### 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 using a helper function during the semantic analysis. This helper function receives a tuple descriptor for the two new types, LAMBDA_TABLE and LAMBDACURSOR and returns another one as input for the table function.

### 6 TABLE FUNCTION IMPLEMENTATIONS

As part of this work, three algorithms have been implemented as table functions to demonstrate and evaluate PostgreSQL with lambda expressions and enhanced JIT compilation: 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.

#### 6.2 PageRank

The PageRank algorithm is a graph mining algorithm that labels nodes according to their incoming and outgoing vertices. Page and Brin [19] developed the algorithm for ranking websites as the foundation of Google’s search engine. 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 the PageRank value, that devolves to connected nodes in each iteration. After convergence, the final PageRank value corresponds to the node’s importance.
• the damping factor $\alpha$ and
• a threshold $t$ for termination of the algorithm.

$T$ describes the input tuples that store information about the vertices. Each lambda expression, $\lambda_{\text{src}}$ and $\lambda_{\text{dst}}$, selects per tuple the input and output node, that may 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 value of all nodes connected via incoming edges with $\alpha$ as the damping factor:

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

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

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

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

The input tuples $T$ are passed to the table function as a LAMBDATABLE parameter. This allows the function to allocate the right amount of memory for the data structures needed for the algorithm before reading the input data. To allow an efficient computation, the edges are stored in a compressed sparse row format similar to Hyper [20]. This also includes 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.

![Figure 10: Edge table and dictionary for dense relabelling.](image)

![Figure 11: Compressed sparse row encoding example: The node with the new key 1 is connected via incoming edges to the nodes 0, 2, 3, 4.](image)

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

Multiple worker threads compute the actual PageRank in parallel. Each of the $n$ worker threads is assigned (at most) $\left\lceil \frac{|P|}{n} \right\rceil$ nodes, for which the next PageRank value should be computed 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 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 holding the final PageRank values. Figure 12 shows an example output of the PageRank function for a person-knows-person data set.

![Figure 12: Example output of the PageRank function.](image)

### 6.3 k-Means Clustering

The third table function implemented as part of this work 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 $\lambda_{\text{dst}}$, 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 $\lambda_{\text{dst}}$ 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 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 will spawn worker threads. For this task, the distance function $\lambda_{\text{dist}}$ is injected directly into the worker thread code, using the thread-safety of the JIT injection framework explained earlier. Each of the $n$ worker processes (at most) $\left\lceil \frac{m}{n} \right\rceil$ points in each iteration. The point-cluster distance is computed for each point/cluster pair and the point is finally assigned to the cluster with minimum distance:

$$c_{i+1}(p) \leftarrow \arg\min_{c \in C} \lambda_{\text{dist}}(p, c). \quad (7)$$

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 benchmark the table functions with 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 terminate after 80 iterations. For PageRank, the damping factor $\alpha$ was set to 0.85, the threshold to 0.00001 and we terminate computation at least after 100 iterations.

#### 7.1 Data sets

The LDBC Social Network Benchmark\(^5\) data set is used for the PageRank evaluation. It simulates activity on a social network and also 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 $1.9 \cdot 10^8$ edges for the person-knows-person relation. The PageRank algorithm without damping is used to determine the most "famous" persons in the database.

An excerpt from the Chicago taxi trip data set\(^6\) is used to evaluate k-Means clustering. It contains all Chicago taxi rides including drop-off location given as latitude and longitude coordinates. An additional data set for the k-Means clustering algorithm consisting of $2 \cdot 10^5$ uniformly distributed Euclidean points in $[-500.0, +500.0]$ was generated directly from an SQL script. These points are also used for evaluating the labelling function.

Also, we test the labelling function with generated data (using the built-in `generated_series` function) of varying size.

#### 7.2 Test Environment

All experiments were run on an 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 was compiled on the machine with LLVM 7 for JIT support. Besides, a HyPer instance supporting the new lambda operators\(^2\) was running on the server for comparative measurements.

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

Most of the measurements have been 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 measured time spans the entire execution time including parsing and compilation time. For measuring 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 evaluate 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
Figure 15: Varying the input size for k-Means with $k = 10$, two dimensions and 80 iterations: (a) with the Chicago taxi data set and (b) with randomly generated points. (c) compares the different compilation modes.

For the label function, both high-performance (L3) and basic JIT-compiled (L2) lambda expressions were tested (see Figure 16). The high-performance implementations are highly competitive with their HyPer counterparts, whereas HyPer outperforms the built-in PostgreSQL JIT compilation for larger 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 the functions were implemented. These allow to vary the number of worker threads 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 on. This conforms with the number of logical cores of the evaluation server and shows that the algorithms use all available cores efficiently. For the Chicago taxi trips data set, the saturation occurs already from 40 threads on due to overall cluster convergence being considerably faster than for uniformly distributed data.

Figure 16: Varying the input size for labelling.

Figure 17: PageRank: LDBC Person-Knows-Person data set, $10^5$ tuples, L2 JIT, $\alpha = 0.85$, 100 iterations.

For k-Means, we evaluate 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, 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 is very low and hardly measurable for smaller inputs.

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

This work 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. When integrating 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 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 can be read from table functions. Furthermore, table functions using one of the new subquery types were equipped with a automatic row type deduction, rendering column definition lists in SQL unnecessary and hence making the use of the new table functions more convenient.

PostgreSQL had already supported basic JIT compilation before, which was the starting point for the new lambda function evaluation. This work introduced four modes for evaluating lambda expressions, two of them use the existing JIT compilation system and the other two use a new JIT compilation framework backed by lightweight LLVM expressions. This idea was partly inspired by the query compilation found in the HyPer database system, which entirely compiles queries into native code before executing them. Three exemplary data mining algorithms, namely, k-Means, labelling and PageRank were implemented as table functions using the new lambda expression system. The evaluation with different types of test data sets showed highly competitive performance and scalability and makes the extension a promising foundation for various data mining tasks in the PostgreSQL database system.

REFERENCES


