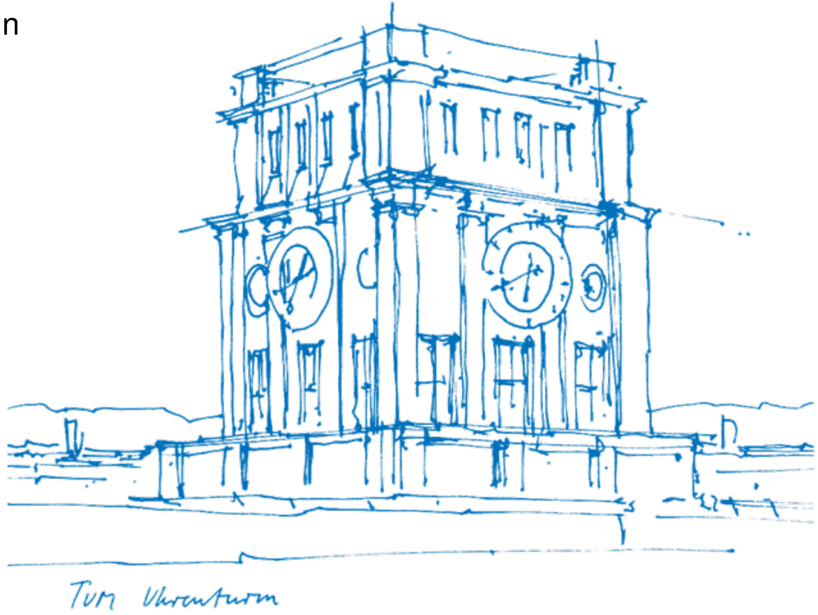
Chair III: Database Systems
Chair XXV: Data Science and Engineering
Department of Informatics
Technical University of Munich

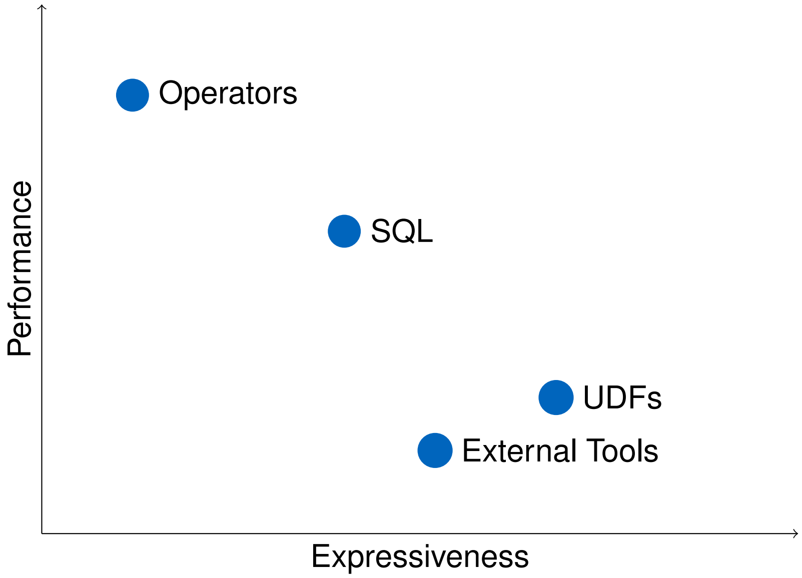
Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL

Maximilian E. Schüle, Jakob Huber, Alfons Kemper, Thomas Neumann

Vienna, Austria, July 7-9, 2020

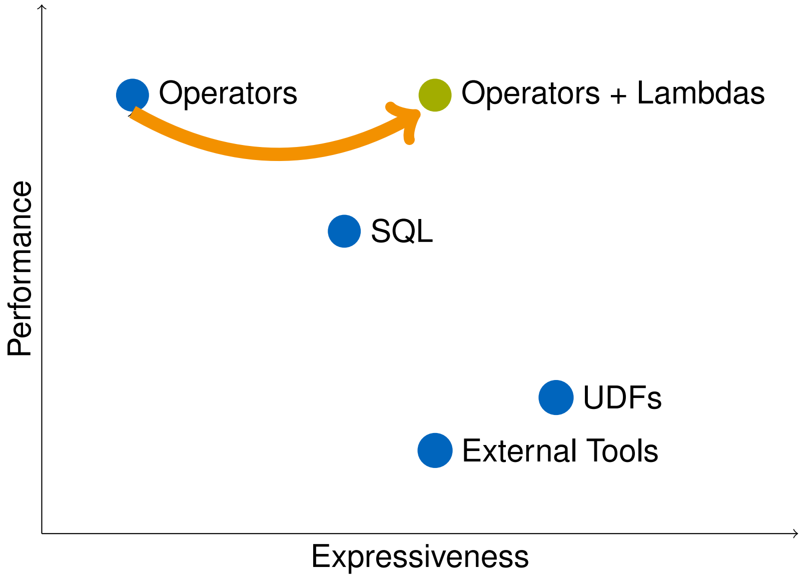


Why Lambda Functions in SQL?



- SQL
  - Turing-complete with recursive tables
  - queries get optimised before execution
  - statements must be expressed in relational algebra
- Operators (Table Functions)
  - purpose-specific but high-performant
  - require development by a database engineer
- User-Defined Functions (UDFs)
  - allow procedural language statements in SQL
  - not as performant as operators
- External Tools
  - database system as storage layer only
  - time consuming extraction necessary

Why Lambda Functions in SQL?



- SQL
  - Turing-complete with recursive tables
  - queries get optimised before execution
  - statements must be expressed in relational algebra
- Operators (Table Functions)
  - purpose-specific but high-performant
  - require development by a database engineer
- User-Defined Functions (UDFs)
  - allow procedural language statements in SQL
  - not as performant as operators
- External Tools
  - database system as storage layer only
  - time consuming extraction necessary
- Operators + Lambdas
  - customisation of operators

Lambda Functions in HyPer

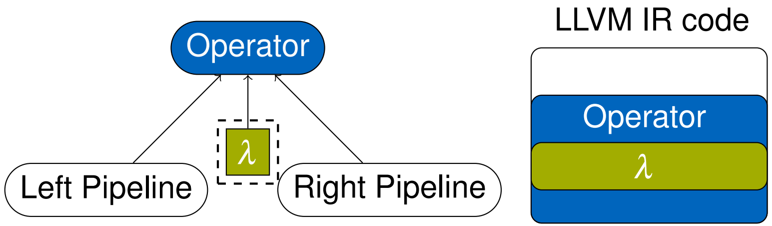
- HyPer: code-generating database system
- produces LLVM IR (Intermediate Representation)
- Lambda expressions: inject code into regular operators
- composed of *lambda arguments* to identify tuples and
- a *lambda body* to formulate an expression

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

- Example: k-Means with injected distance metric

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

- Currently only implemented in HyPer
- Corresponding source-code: restricted




```
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);
```

Challenges when Integrating Lambda Expressions in PSQL

- Support for table arguments
  - table access inside of table functions needed
  - SQL:2016 supports polymorphic table functions
  - but not yet integrated in PostgreSQL
- Support for lambda functions
  - registration as PostgreSQL expression
- Just-in-Time (JIT) code compilation with LLVM
  - supported since PostgreSQL version 11 for expressions
  - type and validity checks slow down performance
  - these checks not needed for lambda expressions
  - do not allow multi-threading with lambda expressions

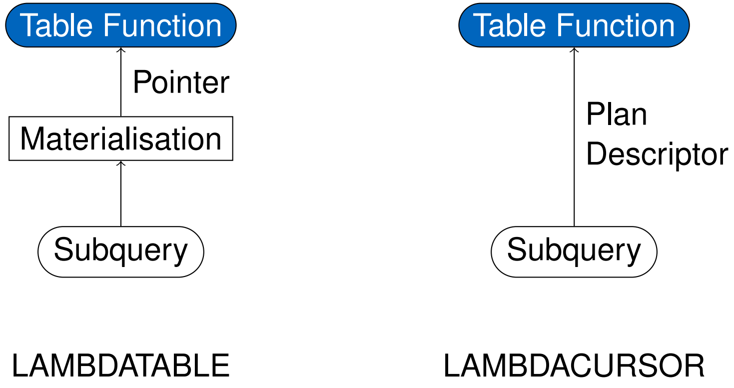
```sql
CREATE FUNCTION mytablefunc(
    TABLE in_tab) AS [...];

-- table function call with table as input
SELECT * FROM mytablefunc(TABLE (<table>))
```

Listing 1: Polymorphic table functions in SQL:2016 (ISO/IEC TR 19075-7).

https://standards.iso.org/ittf/PubliclyAvailableStandards/c069776_ISO_IEC_TR_19075-7_2017.zip

Tables and Lambda Expressions as Subarguments

- Support for table arguments
  - Current approaches (like MADlib): table name as subargument, this requires another database connection to access the data
  - two solutions: `LAMBDATABLE` and `LAMBDACURSOR`
  - `LAMBDATABLE`: materialises the data in a tuplestores
  - `LAMBDACURSOR`: returns a plan descriptor to fetch data tuple-wise
- Support for lambda functions
  - added keyword `LAMBDA`
  - syntax similar to HyPer
  - *lambda arguments* to identify the tuples
  - *lambda body* to express the function




```
LAMBDA(name_1, name_2, ...)(expr).
```

Listing 2: Proposed lambda expression for PSQL.

Modification of the PostgreSQL Engine

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

`Expr Node`: holds information of the lambda expression
Execution Steps:

1. Parser: added rules for lambda expression
2. Analyser: added lambda for type deduction
3. Planner/Optimiser: distinguishes between `LAMBDACURSOR` and `LAMBDATABLE` (separate materialisation)
4. Executer: passes lambda expression to the table function

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

Listing 3: Expr Node: the C struct LambdaExpr.

Usage of Lambda Expressions

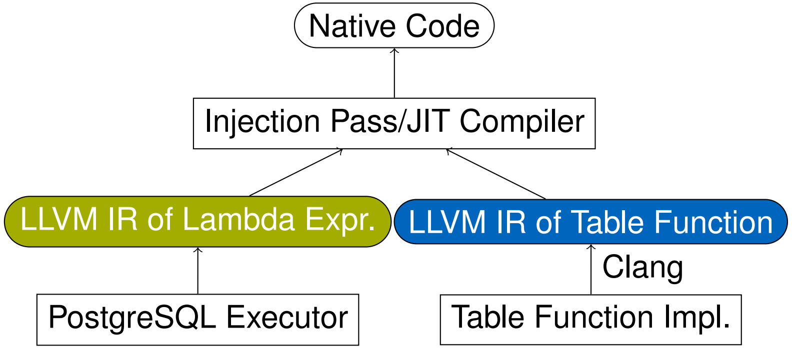
- Table function creation:
  - `LAMBDATABLE` or `LAMBDACURSOR` signals table as input
  - `LAMBDA` indicates the position of the lambda expression
  - access to input tables and lambda expression evaluation happens inside of the shared library written in C
- Table function call:
  - SQL statements for input tables
  - `LAMBDA` expression as defined: lambda arguments identify the tuples, lambda body expresses the function

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

SELECT * FROM foo(
  (SELECT * FROM input1), (SELECT * FROM input2),
  LAMBDA (a)( sqrt(a.x^2 + a.y^2))
);
```

Listing 4: Table function with two input tables and one lambda expression.

Code-Generation for Lambda Functions

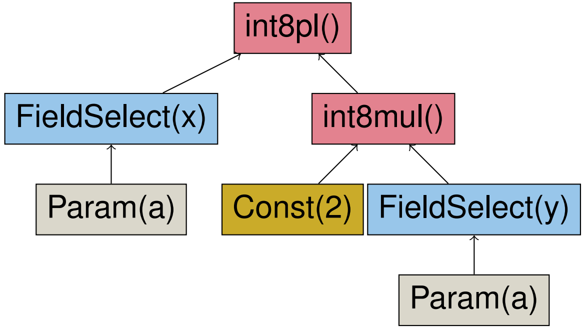
- **Interpreted execution (L1)**: evaluated as an ordinary PostgreSQL expression with a computed `goto` approach
- **JIT-compiled execution (L2)**: using existing JIT optimisations for PostgreSQL expressions
- **High-performance JIT-compiled execution (L3)**: using a custom LLVM wrapper for thread-safe lambda execution
- **High-performance JIT injection (L4)**: same as the previous mode, but the code injected into the table function



LLVM Wrapper for JIT-compilation

- JIT compilation by PostgreSQL: stores result of each Opcode in fixed memory positions
- Stack-based buildup of the LLVM structure to allow multi-threaded execution
- Example for supported Opcodes:

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



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

Implemented Table Functions

PageRank

- calculates the PageRank for nodes given as set of edges
- input arguments: the input table (the edges), parameters
- two lambda expressions, each indicating either source or destination

```
postgres=# SELECT * FROM pagerank(
 (SELECT src,dst FROM knows),
 LAMBDA(src)(src.src), LAMBDA(dst)(dst.dst),
 0.9, 0.001, 45);
```

k-Means

- clusters points to *k* clusters
- input tables: one for the initial clusters, one for all points
- lambda expression defines the distance metric
- returns input points assigned with a cluster number

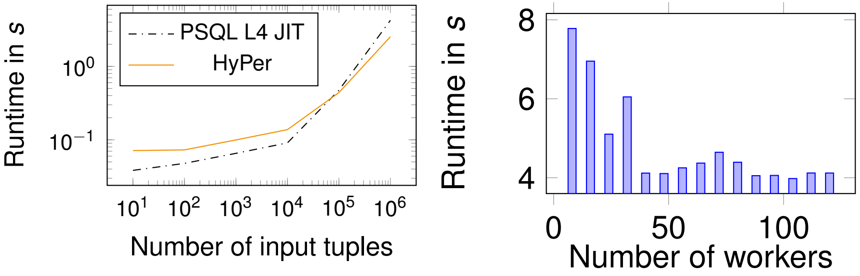
```
postgres=# SELECT * FROM kmeans(
 (SELECT lat, lng, rowid FROM airports LIMIT 8),
 (SELECT lat, lng, rowid from airports),
 LAMBDA(a,b)((a.lat-b.lat)^2+(a.lng-b.lng)^2), 8);
```

Evaluation: Set-Up

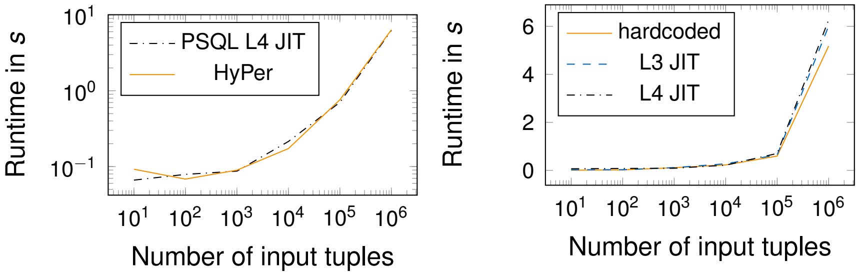
- Ubuntu 18.04 LTS, Intel Xeon CPU E5-2660 v2 processor, 2.20 GHz (20 cores), 256 GiB DDR4 RAM
- PostgreSQL version 11.2 with LLVM 7 for JIT support vs. HyPer
- Five runs per test, results were averaged.
- `work_mem` configuration of PostgreSQL set to 8 GB (working only in main memory)

Evaluation: k-Means

Chicago Taxi Data

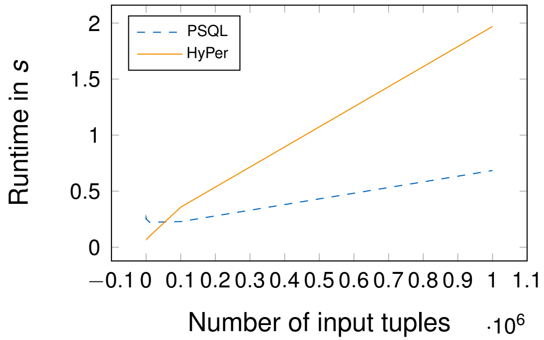


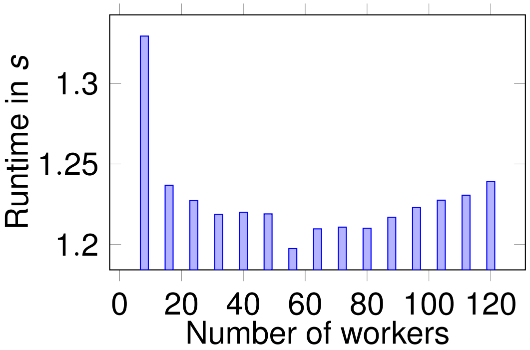
Random Points



- Chicago taxi trip data set (10^6 tuples): clustering on drop-off locations (latitude and longitude)
- Random points: $2 \cdot 10^7$ uniformly distributed Euclidean points in $[-500.0, +500.0]$
- 10 clusters, 80 iterations
- Performance of operators in PostgreSQL similar to those of HyPer, near hard-coded performance in PostgreSQL
- Scales with number of available threads

Evaluation: PageRank





- LDBC Social Network Benchmark: person-know-person relationship
- Scale factor 10 ($1,9 \cdot 10^6$ edges, $\alpha = 0$ (no damping)
- PostgreSQL: constant overhead of about 250 ms, caused during preprocessing when creating a dictionary
- Scales with number of available threads

Conclusion

- Integrated lambda expressions in an open-source database system (PostgreSQL)
- Added support for table arguments in PostgreSQL
- LLVM wrapper for just-in-time compiling lambda expressions
- Exemplary usage with PageRank and k-Means
- Comparable performance to lambda expressions in HyPer
- Future work: address lambda expressions directly in PL/pgSQL

Thank you for your attention!

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