Runtime Specialization of PostgreSQL Query Executor

Eugene Sharygin[1,2(✉)], Ruben Buchatskiy[1], Roman Zhuykov[1], and Arseny Sher[1,2]

[1] Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia
{eush,ruben,zhroma}@ispras.ru, sher-ars@yandex.ru
[2] Lomonosov Moscow State University, Moscow, Russia

Abstract. For computationally intensive workloads, achieving high database performance is in direct correspondence to utilizing CPU efficiently. At the same time, interpretation overhead inherent to traditional interpretive SQL engines gets in the way of optimal CPU utilization.

One solution to this problem is dynamic query compilation, which consists in generating efficient machine code at run time given a particular input query.

Creating a complete query compiler from scratch for an existing database system takes a large amount of development and maintenance effort. Similar results, however, can be obtained more easily using program specialization of a generic query engine with respect to a particular query.

This paper presents intermediate results of applying this approach to the query engine at the core of PostgreSQL database system.

Keywords: Dynamic optimization · JIT compilation
Partial evaluation · Runtime specialization · Query execution
PostgreSQL · LLVM

1 Introduction

One of the central parts of any database system is its query engine, which takes a query from an input channel (usually a network socket) and executes it on the database. This is naturally an interpretive process, and, in fact, in most database systems query engines are implemented as straight query interpreters.

However, interpretation overhead often takes its toll on the overall performance of a database system, which warrants a search for a more efficient query engine implementation.

One usual alternative to interpretation is just-in-time compilation, which has been widely deployed across different domains, including database systems

This work is supported by RFBR grant 17-07-00759 A.

A. K. Petrenko and A. Voronkov (Eds.): PSI 2017, LNCS 10742, pp. 375–386, 2018.
https://doi.org/10.1007/978-3-319-74313-4_27

[CPS+81, Gre99, KVC10, Neu11, NL14, MB17]. Creating a just-in-time compiler takes large amounts of development effort.

Some developing compiler toolchains strive for minimizing the programmer effort needed in order to arrive at an efficient language implementation, for example, by deriving efficient JIT compilers from specially crafted interpreters [WWS+12] or providing tools to develop compilers in the style of interpreters [RO10], all with little to no guidance from language implementors. However, these techniques are not applicable to existing language interpreters.

Micro-specialization [ZDS12, ZSD12a, ZSD12b] is another approach to optimizing DBMS performance, which consists in replacing particular utility functions with implementations specialized at run time based on hand-crafted templates. However, the applicability and efficiency of this approach is limited due to the lack of automaticity.

For the task of developing a query compiler in an extension to an existing database system, ideally, we would like to keep the source code of its existing interpretive query engine as a reference implementation of the database business logic instead of creating a parallel but different implementation. Given the source code of an interpreter and a new query, the tool would produce efficient machine code that is semantically equivalent to the interpreter, but has superior performance. This would allow to combine interpretive and compiled execution within a single database system and continue to use the interpreter for queries that are not worth the compilation effort. Perhaps more importantly, this would redeem us from the burden of continuing support and maintainance of our newly developed just-in-time compiler as to bringing new features and bug fixes from the existing query interpreter.

Automatic program specialization is exactly that tool. Generally, it is a type of optimizing program transformation that, given a program and some of its input data, generates a residual program that, when given remaining input data, yields the same result as running the original program on all of its input data.

This paper describes the ongoing work in developing a specializer for PostgreSQL [Pos] query executor using LLVM compiler infrastructure [LLVM]. It is organized as follows. Sections 2 and 3 describe online and offline partial evaluation, which are two different specialization methods that we used. Section 4 describes application of runtime specialization to PostgreSQL query engine. Finally, Sect. 5 concludes the paper.

2 Online Partial Evaluation for LLVM

Online partial evaluation is the kind of specialization that is based on aggressive constant propagation and loop unrolling.

Online partial evaluator takes a root function and a list of values of the arguments and starts repeatedly evaluating the function's basic blocks until reaching a fixed point. Partial evaluator attempts to evaluate each instruction statically given known values of its operands. If it succeeds in doing so, the instruction is subsequently eliminated and its value is replaced with the constant it has been

evaluated to. This basic procedure can be implemented rather efficiently using the properties of the SSA form which is provided by LLVM.

In contrast to compile-time online partial evaluators such as LLPE [Smo14], a runtime partial evaluator can take advantage of the constant values residing in the process memory. This is implemented by the following transfer function:

$$f_{x=\mathrm{load}(a)}(m) = \begin{cases} m[x \mapsto \mathrm{heap}(m(a))], & m(a) \in \mathit{Const}_0 \\ m[x \mapsto \bot], & m(a) = \bot \\ m[x \mapsto \top], & m(a) \notin \mathit{Const}_0 \cup \{\bot\} \end{cases},$$

where $m\colon \mathit{Var} \to \mathit{Const} \cup \{\top, \bot\}$ is a lattice value, $\mathit{Const}_0 \subseteq \mathit{Const}$ is a set of addresses of constants in the heap, and $\mathrm{heap}(c)$ is the value of the memory cell at the given location.

Despite the conceptual simplicity, we faced several problems with online partial evaluation which mostly boil down to the following:

- The results are hard to test, debug, and visualize. Online partial evaluation fuses annotation and specialization into a single run-time phase, which (1) requires a developer to find test queries exercising very specific parts of the code in order to ensure the expected binding-time division and (2) provides no intermediate representation better suited for analysis and visualization than specializer-generated code with its lack of static parts, duplicated dynamic code and unrolled loop iterations.
- Having no means to conduct the analyses ahead-of-time means that continuing development of online partial evaluator inevitably leads to trading complexity and accuracy of partial evaluation for reduced run-time overhead (or vice versa), which is highly suboptimal because the binding-time division does not have to be done at run time since it doesn't require anything but the source code of the interpreter.

3 Offline Partial Evaluation for LLVM

In contrast to online partial evaluation, offline partial evaluation proceeds in two distinct phases:

1. At compile time, binding-time analysis (BTA; see Sect. 3.2) takes the root function (the entry point of the interpreter) and the initial argument division, which is a list of binding times ("static" or "dynamic") per each of its arguments. The values of static arguments are known at specialization time (and not during BTA), while the values of dynamic arguments are only known at query execution time (and not at specialization time). BTA then annotates each instruction as either static or dynamic, which indicates whether the instruction can be completely evaluated at specialization time or not.
2. The annotated program is then specialized at run time (Sect. 3.3) with respect to concrete values of static arguments, which results in a program specialized to those values. Specialization combines execution of static program fragments with generating residual code for dynamic program fragments.

One of the main benefits of offline partial evaluation is the ability to separate these two phases in time: BTA can be conducted ahead-of-time (which makes it a lot easier to test and debug the BTA, and visualize the results), while specialization (directed by annotations in a rather straightforward manner) runs just-in-time.

To our knowledge, this is the first work to describe in detail and implement such a scheme for LLVM IR, although the idea has been proposed before [LC14].

3.1 Representation of Binding Times in LLVM IR

As for the intermediate representation BTA and specialization are performed at, we find LLVM IR rather satisfying. It allows a developer to use the wide range of tools, either included in LLVM or external, and to integrate program specialization gradually into an otherwise LLVM IR-centric system (such as our traditional JIT-compiler for PostgreSQL we are basing this work upon [MB17]).

```
define i32 @power.ds(i32 %x, i32 %n) !arg_spec !1 {
entry:
  %n.mod2 = srem i32 %n, 2, !static !2
  %n.odd = icmp eq i32 %n.mod2, 1, !static !2
  br i1 %n.odd, label %odd, label %even

odd:
  %n.1 = phi i32 [ %n, %entry ], [ %n.half, %even.step ], !static !2
  %x.1 = phi i32 [ %x, %entry ], [ %x.sqr, %even.step ]
  %n.dec = add nsw i32 %n.1, -1, !static !2
  %result.prev = tail call i32 @power.ds(i32 %x.1, i32 %n.dec)
  %result.odd = mul nsw i32 %result.prev, %x.1
  ret i32 %result.odd

even:
  %n.2 = phi i32 [ %n.half, %even.step ], [ %n, %entry ], !static !2
  %x.2 = phi i32 [ %x.sqr, %even.step ], [ %x, %entry ]
  %n.positive = icmp sgt i32 %n.2, 0, !static !2
  br i1 %n.positive, label %even.step, label %zero

even.step:
  %x.sqr = mul nsw i32 %x.2, %x.2
  %n.half = lshr i32 %n.2, 1, !static !2
  %n.half.mod2 = and i32 %n.half, 1, !static !2
  %n.half.even = icmp eq i32 %n.half.mod2, 0, !static !2
  br i1 %n.half.even, label %even, label %odd

zero:
  ret i32 1
}
```

Fig. 1. Example of LLVM IR with binding-time annotations

LLVM IR provides a means of expressing domain-specific semantics using the notion of a metadata: each instruction or function can be labeled in an application-specific way, which is exactly what we need to represent binding-time annotations and to link the annotated code of a particular function to its source (so that it can later be discovered by the specializer). Figure 1 shows an example of binding-time annotations as expressed using LLVM IR metadata.

Overall, our BTA operates on the LLVM module containing the source code of the interpreter, and, one by one, annotates each of its functions, in effect creating annotated function variants and linking them to their sources. The result is a fully functional LLVM module, which contains both all the original functions and their annotated variants.

3.2 Binding-Time Analysis

BTA described here is polyvariant in functions and monovariant in basic blocks, meaning that it can produce multiple annotated variants for a single function (effectively cloning each function for each group of contexts with matching argument divisions) but it does not clone function's basic blocks or annotate instructions within a single function ambiguously.

For any given function, BTA computes binding-time annotations for each of its instructions (except for control flow transfer instructions, see below). Any instruction's binding time is a function of the binding times of its operands.

The basic rule, called the congruence condition, that drives the analysis is that all instruction users (with the sole exception of the call instruction — see below) of a dynamic value are themselves dynamic by necessity. In order to compute the solution preserving the maximum amount of static information, the algorithm starts from annotating most of the instructions as static and then repeatedly fixes binding times by restoring this property across SSA edges (by changing some binding times back to dynamic), until reaching the fixed point.

Annotating Memory Access Instructions. In addition to the binding time of the address operand, annotating a memory access instruction also needs the binding-time type of a corresponding data type, which is a supplementary piece of annotation indicating which fields of a data structure, if any, can be loaded at specialization time. If both the address operand and the accessed field are static, then the load is safe to be annotated static.

```
%ExprState = type {
  static i32,  ;tag
  static i8,  ;flags
  dynamic i8,  ;resnull
  dynamic i64,  ;resvalue
  static %TupleTableSlot*,  ;resultslot
  static %ExprEvalStep*,  ;steps
  static %Expr*   ;expr
}
```

Fig. 2. Binding-time type annotations

Binding-time types are expressed in a dialect of LLVM IR (not in LLVM IR proper because the latter does not currently allow metadata on types). See example in Fig. 2.

The reason these annotations need to be supplied rather than computed is not only predictability and simplicity of the approach, but mainly that the latter would require a whole-program alias analysis which would lead to overly conservative results since the query interpreter we apply the specializer to performs multiple calls to external database management functions which are out of scope of specialization.

Annotating Control Transfer Instructions. Control flow transfer instructions such as branch or return are always annotated dynamic for the reason that doing the opposite would effectively mean combining several blocks into one at specialization time, which is not only of little value because the post-specialization code is optimized by LLVM anyway, but also actively harmful since it can lead to needless code duplication and even non-termination (resulting from, for example, duplicating instructions in the loop header block when statically branching to it from the latch).

Annotating Loops. The way the specializer does loop unrolling is by cloning the body of a loop per each set of values of variables reaching its header (see Sect. 3.3). If any of such variables are static and not constants, but the loop is controlled by a dynamic condition, then the specializer won't terminate.

```
loop:
  %n = phi i32 [ 0, %0 ], [ %n.inc, %loop ], !static !0
  %n.inc = add i32 %n, 1, !static !0
  %cmp = icmp slt i32 %n, %d
  br i1 %cmp, label %loop, label %exit
```

Fig. 3. Static variables in the header of a dynamic loop

The problem is illustrated by Fig. 3. On each subsequent iteration of this loop, the values of static variables `%n` and `%n.inc` will be different: $(0, 1)$, $(1, 2)$, $(2, 3)$ and so on, and the specializer won't terminate in an attempt to unrolling it.

The solution is to check if exit conditions in the latch blocks of a loop are dynamic, and if this is the case, annotate all changing variables in the header of the loop as dynamic.

Annotating Function Calls. Function calls are an exception to the congruence condition because a call can be annotated static even if some of its arguments are dynamic. The reason is that the semantics of a call instruction being static is different from that of other instructions: static calls are still performed at run time (unless the function is completely static (see Sect. 3.3) — but this is nothing but a minor optimization, and nothing is conceptually changed in what follows), the binding time simply indicates that the return value is known at specialization time, before the call has to be made at run time.

The binding time of a call instruction is hence determined by the binding times of values returned from the called function when the latter is analyzed according to the argument division at the call site. Therefore, annotating a function call requires performing binding-time analysis for a called function. On the other hand, the binding time of a call instruction may influence the binding time of a value returned from the function as well. In case of recursive function calls, this is a cyclic dependency that needs to be resolved.

The simplest way to resolve this is to initialize return binding times for all functions as dynamic and use this default value for all calls which are back edges in the call graph. This has an obvious downside of failing to handle static recursion as such — but the latter is barely used in PostgreSQL source code which is our primary application (see Sect. 4).

The algorithm that we use for analyzing a single function is thus as follows:

1. Initially, annotate all function results as dynamic and all call instructions as static.
2. Upon reaching a fixed point in annotating the individual instructions, recursively analyze all called functions that are not already being analyzed, according to the argument divisions at corresponding call sites. Reannotate call instructions as dynamic if return binding times of corresponding functions are dynamic.
3. If binding time of at least one of the calls has changed, analyze the function again until the fixed point is reached, and repeat the algorithm.
4. If no binding times have changed, change callees appropriately (to refer to the newly annotated functions) and update the binding time of the return value of this function.

Determining binding times of call instructions requires recursively analyzing called functions, perhaps multiple times with different argument divisions, generating annotated functions which won't necessarily be used in the end. This may lead to unpredicatable performance of the BTA phase as a whole, although its termination is guaranteed (see below). This is not, however, a major problem for runtime specialization since BTA is expected to be performed only once.

Termination. Here, we briefly show that binding-time analysis as described in this section always terminates.

Since no function can be annotated twice with regards to a particular argument division and since there are only finitely many function—argument division pairs in any given source program, BTA always terminates as long as annotating a particular function with respect to a particular argument division terminates.

In the course of annotating instructions in a particular function, binding time of any particular instruction never changes from dynamic to static. For calls, this is true because binding time of a function result can't ever change from dynamic to static in case binding time of a particular argument changes from static to dynamic. Since for any particular function annotation only finitely many $S \rightarrow D$ binding-time changes are possible, annotating a particular function always terminates.

3.3 Specialization

Specialization is driven by binding-time annotations produced in the BTA phase. As described here, it is polyvariant both in functions and basic blocks, meaning that for each new pair of an annotated function and a list of static argument values, a new residual function is constructed, and for each new pair of a basic block and a static store containing values of reaching definitions, a new residual basic block is constructed.

Function specialization starts from the entry basic block of a function and the initial static store containing only the values of static arguments in the function call, and proceeds by repeatedly evaluating basic blocks and their successors and updating corresponding static stores, until there are no more pairs of (block, store) to evaluate. This strategy effectively propagates particular static values towards their uses by duplicating conditional branches and unrolling loops.

Any particular annotated basic block can be evaluated multiple times, but each time a new residual basic block is constructed, a new set of residual definitions is built from the same set of dynamic instructions in the annotated code, thus maintaining the SSA property automatically.

Result of specializing function `@power.ds` (Fig. 1) with respect to `%n=4` is shown in Fig. 4.

```
define i32 @power.4(i32 %x) {
  %x.sqr = mul i32 %x, %x
  %x.sqr.1 = mul i32 %x.sqr, %x.sqr
  ret i32 %x.sqr.1
}
```

Fig. 4. Result of specialization for example in Fig. 1

Static Store. Static store is a data structure that maps static variable definitions to particular constant values. Each definition can have multiple values due to the polyvariance of specialization in basic blocks: each basic block, for example in a body of a static loop, can be evaluated multiple times, each time resulting in a new residual basic block.

Static store is organized as a tree of scopes. At any point during specialization process, the state of the specializer includes some particular leaf node in the store, and new leaf nodes can be added when a control flow edge is visited. Any particular leaf node is associated with a particular residual basic block, and every node on the path to the root scope corresponds to some block that dominates the current block in the CFG — in a way that every dominator has at least one scope associated with it, and each pair of basic blocks that corresponds to a pair of scopes immediately following one another in the tree, is itself connected by a control-flow edge.

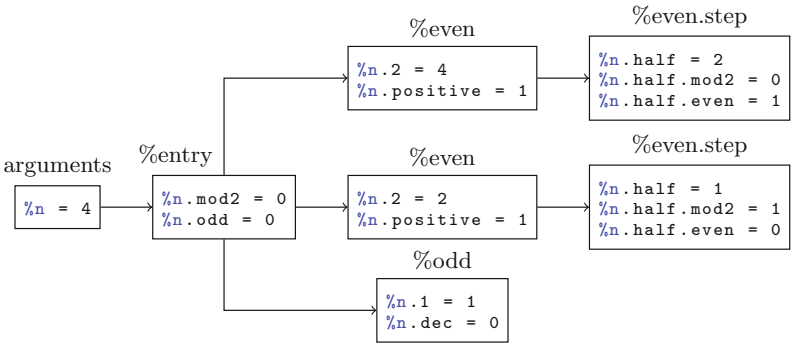

Fig. 5. Static store example

Example in Fig. 5 shows the state of the static store created during specializing the function `@power.ds` (Fig. 1) to `%n=4` (see the residual code in Fig. 4). The root scope of the store contains the value for the static argument `%n`. During specialization, the basic blocks `%even` and `%even.step` are visited twice each, and so two pairs of scopes are created.

Static Functions. If all non-terminating instructions in a function are annotated as static, then the function represents a completely static calculation and does not need a residual function. In fact, if such a function was to be constructed at specialization time, it would contain nothing but a single return instruction with a static value.

As an optimization, these cases are annotated appropriately at binding-time analysis time, and no residual function is created at specialization time — instead, the specializer evaluates the entirety of the function statically, and replaces corresponding calls with a single static result.

4 Runtime Specialization of PostgreSQL Query Executor

We finally describe the application of program specialization to PostgreSQL query executor.

Query execution in PostgreSQL is implemented in several stages: parsing and rewriting, which manipulate a query's syntactic representation, optimization, which constructs a plan tree, and execution, which interprets a plan tree (using Volcano (also known as pull-based) execution model [G94]) in order to obtain the result.

PostgreSQL provides hooks for extensions into almost all of these stages. In particular, `ExecutorRun` hook provides the ability to replace the default execution strategy (plan interpretation) with query compilation.

Given a particular SQL query, our specialization-based query compiler specializes PostgreSQL source code on the fly in order to obtain efficient machine code. Overall, the method can be summarized by the following:

1. At compilation time (only for offline partial evaluation):
   (a) First, PostgreSQL source code is compiled with Clang, which results in an LLVM IR module containing a function named `ExecutePlan`, which is an entry point to the built-in query interpreter. This function takes as arguments a query plan and an execution context and evaluates the plan according to the context.
   (b) The function `ExecutePlan` is then annotated according to its binding-time division, which results in a new function $\texttt{ExecutePlan}^{\texttt{ann}}$. All direct and indirect callees are annotated as well.
2. At run time:
   (a) The function $\texttt{ExecutePlan}^{\texttt{ann}}$ is specialized with respect to a particular query plan and execution context at hand, resulting in $\texttt{ExecutePlan}^{\texttt{res}}$.
   (b) The function $\texttt{ExecutePlan}^{\texttt{res}}$ is further optimized and compiled by LLVM JIT to machine code, which is then immediately executed.

Along with the query-executor-related parts of PostgreSQL source code, binding-time analysis for offline partial evaluation also takes as input the LLVM IR file with binding-time type definitions of struct types used when annotating memory operations (see Sect. 3.2). Notable examples of such struct types are query plan nodes and expression nodes.

Online partial evaluation does not have a compile-time phase, but instead requires a run-time preprocessing phase to gather all memory addresses of constants in the heap (in order to define the heap() function — see Sect. 2).

5 Conclusion

In this paper we described program specialization methods and their application to PostgreSQL query executor.

We are developing prototype query compilers based on online (Sect. 2) and offline (Sect. 3) partial evaluation. Online specializer is implemented for PostgreSQL 9.6 and shows up to 1.4x speedup on some synthetic queries. Offline specializer is implemented for PostgreSQL 10 and shows up to 1.4x speedup on TPC-H Q1 which is part of the industry-standard TPC-H benchmark [TPC-H]. Offline specializer currently requires some minor binding-time improvements applied to the PostgreSQL codebase.

Our experiments show that runtime specialization can be used to effectively eliminate interpretation overhead and inline static query and database parameters into the compiled machine code, but its performance is not currently on par with that of the traditional query compiler [MB17] that we are developing separately, which shows up to 5.5x speedup on Q1. We speculate that this difference is due to push-based execution model that the compiler implements and other algorithmic improvements that can't be automated.

We propose implementing algorithmic improvements separately in C and applying runtime specialization on top. We started by implementing the push model for PostgreSQL. Combined with the offline specializer, it shows 1.6x speedup on TPC-H Q1 compared to PostgreSQL 10.

Our implementation also suggests that query compilers can effectively combine specialization of some parts of an interpretive query engine with traditional compiled implementation of the other in order to maximize efficiency of generated code.

References

[CPS+81] Chamberlin, D.D., Putzolu, F., Selinger, P.G., Schkolnick, M., Slutz, D.R., Traiger, I.L., Yost, R.A.: A history and evaluation of System R. Commun. ACM **24**(10), 632–646 (1981). https://doi.org/10.1145/358769.358784

[G94] Graefe, G.: Volcano - an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6**(1), 120–135 (1994). https://doi.org/10.1109/69.273032

[Gre99] Greer, R.: Daytona and the fourth-generation language cymbal. Sigmod, 525–526 (1999). https://doi.org/10.1145/304181.304242

[KVC10] Krikellas, K., Viglas, S.D., Cintra, M.: Generating code for holistic query evaluation. In: Proceedings of International Conference on Data Engineering, pp. 613–624 (2010). https://doi.org/10.1109/ICDE.2010.5447892

[LC14] Lomuller, V., Charles, H.-P.: A LLVM extension for the generation of low overhead runtime program specializer. In: Proceedings of International Workshop on Adaptive Self-Tuning Computing Systems - ADAPT 2014, pp. 14–16 (2014). https://doi.org/10.1145/2553062.2553064

[LLVM] The LLVM Compiler Infrastructure. http://llvm.org/

[MB17] Melnik, D., Buchatskiy, R., Zhuykov, R., Sharygin, E.: JIT-compiling SQL queries in PostgreSQL using LLVM. Presented at PGCon 2017 (2017). http://www.pgcon.org/2017/schedule/events/1092.en.html

[Neu11] Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. **4**(9), 539–550 (2011). https://doi.org/10.14778/2002938.2002940

[NL14] Neumann, T., Leis, V.: Compiling database queries into machine code. IEEE Data Eng. Bull. **37**(1), 3–11 (2014)

[Pos] PostgresSQL Global Development Group. PostgresSQL. http://www.postgresql.org/

[RO10] Rompf, T., Odersky, M.: Lightweight modular staging. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering - GPCE 2010, p. 127 (2010). https://doi.org/10.1145/1868294.1868314

[Smo14] Smowton, C.S.F.: I/O Optimisation and elimination via partial evaluation. University of Cambridge, Computer Laboratory, Ph.D. thesis, (UCAM-CL-TR-865), 1131 (2014)

[TPC-H] TPC-H, an ad-hoc, decision support benchmark. Transaction Processing Performance Council. http://www.tpc.org/tpch

[WWS+12] Wurthinger, T., Wob, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C.: Self-Optimizing AST Interpreters (2012)

[ZDS12] Zhang, R., Debray, S., Snodgrass, R.T.: Micro-specialization: dynamic code specialization of database management systems. In: International Symposium on Code 6373 (2012). https://doi.org/10.1145/2259016.2259025

[ZSD12a] Zhang, R., Snodgrass, R.T., Debray, S.: Micro-specialization in DBMSes. In: Proceedings - International Conference on Data Engineering, pp. 690–701 (2012). https://doi.org/10.1109/ICDE.2012.110

[ZSD12b] Zhang, R., Snodgrass, R.T., Debray, S.: Application of micro-specialization to query evaluation operators. In: Proceedings - 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012, pp. 315–321 (2012). https://doi.org/10.1109/ICDEW.2012.43