Micro-Specialization in DBMSes

Rui Zhang [1], Richard T. Snodgrass [2], and Saumya Debray [3]

Department of Computer Science, The University of Arizona

[1]ruizhang@cs.arizona.edu [2]rts@cs.arizona.edu [3]debray@cs.arizona.edu

***Abstract*—Relational database management systems are general in the sense that they can handle arbitrary schemas, queries, and modifications; this generality is implemented using runtime metadata lookups and tests that ensure that control is channelled to the appropriate code in all cases. Unfortunately, these lookups and tests are carried out even when information is available that renders some of these operations superfluous, leading to unnecessary runtime overheads. This paper introduces *micro-specialization*, an approach that uses relation- and query-specific information to specialize the DBMS code at runtime and thereby eliminate some of these overheads. We develop a taxonomy of approaches and specialization times and propose a general architecture that isolates most of the creation and execution of the specialized code sequences in a separate DBMS-independent module. Through three illustrative types of micro-specializations applied to PostgreSQL, we show that this approach requires minimal changes to a DBMS and can improve the performance simultaneously across a wide range of queries, modifications, and bulk-loading, in terms of storage, CPU usage, and I/O time of the TPC-H and TPC-C benchmarks.**

I. Introduction

Relational database management systems are by their nature general, in that they can handle whatever schema the user specifies and whatever query or modification is presented to them. Relational operators work on essentially any relation and must contend with predicates specified on any attribute of the underlying relations. Through such innovations as effective indexing structures, innovative concurrency control mechanisms, and sophisticated query optimization strategies, relational DBMSes are very efficient. Such generality and efficiency has enabled their proliferation and use in many domains.

This generality presents challenges to further increases in performance. Consider accessing attributes of a tuple. Such access requires consulting metadata. The catalog, which contains the schema, must be scanned for each attribute value of the tuple in this relation extracted. Although this catalog lookup has been optimized, its overhead will still accumulate over large relations, representing significant overhead.

DBMS *specialization* is an approach to improving the efficiency of DBMSes by providing a version customized in a way that avoids the inefficiencies resulting from the generality. Much of the work over the last three decades in improving DBMS performance can be characterized as specialization with various levels of granularity. At the *architectural level*, the overall architecture of the DBMS is adapted to better support a class of applications. Examples include column-oriented stores for OLAP, H-store for OLTP, and stream processing DBMSes. At the *component level*, a component oriented to a particular kind of data is added to the DBMS. Examples include new types of operators, indexes, and locking modes. At the *user-stated level*, users write triggers and user-defined functions to achieve better performance. The drawbacks of these three levels of specialization are lack of general applicability for architectural specialization, greater complexity in query optimization, DBMS development, testing, and maintenance for component specialization, and the need for complex user involvement for user-stated specialization.

This paper introduces *micro-specialization*, an approach that applies at a finer granularity than that of architectural or component specialization. Our approach takes advantage of information specific to the particular environment of a DBMS by identifying variables within an individual component whose values—typically, schema metadata or query-specific constants—are invariant *within the query evaluation loop*. This information is used for fine-grained specialization that eliminates unnecessary operations along frequently-taken execution paths, leading to optimized code that is both smaller and faster. Often this loop is evaluated for every tuple in the underlying relation(s), thereby offering the possibility of significant performance improvements. However, since the invariants used for specialization are available only at runtime, such specialization cannot be carried out using traditional compiler techniques: micro-specialization applies at DBMS runtime. This implies that the specialization process itself has to be extremely lightweight, which raises a host of nontrivial technical challenges.

This paper describes DBMS micro-specialization, applies this concept to a complex DBMS, and evaluates its effectiveness and cost. Our contributions are the following.

- We introduce a taxonomy of micro-specialization opportunities, based on the kind of variable(s) inducing the specialization and when the specialization is done.
- We implemented a general module that supports and realizes micro-specialization and added calls to this API within PostgreSQL [14] to apply all three kinds of micro-specialization. The changes that are required to incorporating this module to PostgreSQL are minimal.
- We evaluated our ideas empirically using this prototype implementation, focusing on the TPC-H [27] and TPC-C [26] benchmarks. Our experiments show that each micro-specialization improves performance and that, in concert, micro-specialization can significantly reduce DBMS run times across bulk-loading, complex analytic queries, and random modifications. For TPC-H, per-query execution times improve by up to 33%, with an average

improvement across all queries of over 12%. For TPC-C we observed throughput improvements exceeding 11% (with queries and modifications weighted equally).

Micro-specialization incurs none of the disadvantages of the coarser-grained specializations. Since the DBMS architecture is not changed, it does not constrain the breadth of applications that can be supported. As micro-specialization adds no additional components, it does not increase DBMS complexity. Micro-specialization requires no user involvement. Moreover, micro-specialization has the potential of being applied in concert with the other three forms of specialization. For example, it can be applied directly to column-oriented DBMSes and main-memory-based DBMSes and to new kinds of operators.

We first examine in detail a single micro-specialization that improves the performance of even simple queries. In this case study, we examine the specific code changes, predict the performance improvement, and then validate our prediction with an experiment. Section III examines micro-specialization opportunities broadly with a taxonomy showing where and when to apply micro-specialization. We then introduce a general API and discuss how to insert calls to that API to effect micro-specialization. We apply all three kinds of micro-specialization to PostgreSQL. We then characterize through a set of experiments on the TPC-H and the TPC-C benchmarks the salutary effect of micro-specialization. Section VII places micro-specialization in the broader context of DBMS specialization and compares our approach with some specific approaches that share some qualities with micro-specialization.

II. Case Study

In a DBMS, there are many variables which can in fact be invariant (constant) within the query evaluation loop. For instance, once the schema of a relation is defined, the number of attributes is a constant. Moreover, the type of each attribute, the length of each fixed-length attribute, as well as the offsets of some attributes (those not preceded by a variable-length attribute) are constants for this relation.

Listing 1 excerpts a function, `slot_deform_tuple()`, from the source code of PostgreSQL. This function is executed whenever a tuple is fetched; it extracts values from a stored tuple into an array of `long` integers. The function relies on a loop (starting on line 11) to extract each attribute. For each attribute, a path in the code sequence (from line 12 to line 43) is executed to convert the attribute's value within the stored bytes of the tuple into a long integer (that is, bytes, shorts, and ints are cast to longs and strings are cast to pointers). The catalog information for each attribute is stored in a struct named `thisatt`. As Listing 1 shows, attribute length (`attlen`), attribute physical storage alignment (`attalign`), and attribute offset (`attcacheoff`) all participate in selecting a particular execution path.

Within a conventional DBMS implementation, these variables are used in condition checking because the values of these variables depend on the specific relation being queried. Such generality provides opportunities for performance improvement. Micro-specialization focuses on such variables;

```
1  void slot_deform_tuple(TupleTableSlot *slot, int natts) {
2    ...
3    if (attnum == 0) {
4      off = 0;
5      slow = false;
6    } else {
7      off = slot->tts_off;
8      slow = slot->tts_slow;
9    }
10   tp = (char *) tup + tup->t_hoff;
11   for (; attnum < natts; attnum++) {
12     Form_pg_attribute thisatt = att[attnum];
13     if (hasnulls && att_isnull(attnum, bp)) {
14       values[attnum] = (Datum) 0;
15       isnull[attnum] = true;
16       slow = true;
17       continue;
18     }
19     isnull[attnum] = false;
20     if (!slow && thisatt->attcacheoff >= 0) {
21       off = thisatt->attcacheoff;
22     } else if (thisatt->attlen == -1) {
23       if (!slow && off == att_align_nominal(off, thisatt->attalign)) {
24         thisatt->attcacheoff = off;
25       } else {
26         if (!slow && off == att_align_nominal(off, thisatt->attalign)) {
27           thisatt->attcacheoff = off;
28         } else {
29           off = att_align_pointer(off, thisatt->attalign, -1, tp + off);
30           slow = true;
31         }
32       } else {
33         off = att_align_nominal(off, thisatt->attalign);
34         if (!slow)
35             thisatt->attcacheoff = off;
36       }
37       values[attnum] = fetchatt(thisatt, tp + off);
38       off = att_addlength_pointer(off, thisatt->attlen, tp + off);
39       if (thisatt->attlen <= 0)
40           slow = true;
41     }
42     ...
43   }
44 }
```

Listing 1. The `slot_deform_tuple()` Function

when they are constant within the query evaluation loop, the corresponding code sequence can be dramatically shortened.

We utilize the `orders` relation from the TPC-H benchmark as an example to illustrate the application of micro-specialization. To specialize the `slot_deform_tuple()` function for the `orders` relation, we first identify the variables that are constants. According to the schema, no null values are allowed for this relation. Therefore the null checking statements from lines 13 to 18 are not needed. Instead, we can assign the entire `isnull` array to `false` at the beginning of the function. Since each value of the `isnull` array is a byte, we can collapse the assignments with a few type casts. For instance, the eight assignments of `isnull[0]` to `isnull[7]` can be converted to a single, very efficient statement: `(long*)isnull = 0;`

As discussed earlier, some of the variables in Listing 1 are constant for any particular relation. For the `orders` relation, the value of the `natts` (number of attributes) variable is 9. We apply *loop unrolling* to avoid the condition checking and the the loop-counter increment instructions in the `for` statement. The resulting program simply has nine assignment statements.

```
values[0] = ...;
values[1] = ...;
      ...
values[8] = ...;
```

```
1 void GetColumnsToLongs(char* data, int* start_att, int* offset,
2                         bool* isnull, Datum* values) {
3   *(long*)isnull = 0;
4   isnull[8] = 0;
5   values[0] = *(int*)data;
6   values[1] = *(int*)(data + 4);
7   values[2] = (long)(address + bee_id * 32 + 1000);
8   *start_att = 3;
9   if (end_att < 4) return;
10  *offset = 8;
11  if (*offset != (((long)(*offset) + 3) & ~((long)3)))
12    if (!(*(char*)(data + *offset)))
13        *offset = (long)(*offset + 3) & ~(long)3;
14  values[3] = (long)(data + *offset);
15  *offset += VARSIZE_ANY(data + *offset);
16  *offset = ((long)(*offset) + 3) & ~((long)3);
17  values[4] = (*(long*)(data + *offset)) & 0xffffffff;
18  *offset += 4;
19  values[5] = (long)(address + bee_id * 32 + 1001);
20  *start_att = 6;
21  if (end_att < 7) return;
22  if (!(*(char*)(data + *offset)))
23      *offset = (long)(*offset + 3) & ~(long)3;
24  values[6] = (long)(data + *offset);
25  *offset += VARSIZE_ANY(data + *offset);
26  values[7] = *(int*)(address + bee_id * 32 + 1002);
27  if (!(*(char*)(data + *offset)))
28      *offset = (long)(*offset + 3) & ~(long)3;
29  values[8] = (long)(data + *offset);
30  *start_att = 9;
31 }
```

Listing 2. The Micro-Specialized `GetColumnsToLongs()` Function

Now let's focus on the type-specific attribute extraction statements. The first attribute of the `orders` relation is an four-byte integer. Therefore, we don't need to consult the `attlen` variable with a condition statement. Instead, we directly assign an integer value from the tuple with this statement.

```
values[0] = *(int*)(data);
```

Note that the `data` variable is a byte array in which the physical tuple is stored. Since the second attribute is also an integer, the same statement also applies. Given that the length of the first attribute is four bytes, we add four to `data` as the offset of the second attribute.

```
values[1] = *(int*)(data + 4);
```

The resulting specialized code for the `orders` relation is presented in Listing 2. Although the code looks longer, the `for` loop in Listing 1 has been unrolled nine times. As as result, the specialized code will execute many fewer instructions than the stock code. Manual examination of the executable object code found that that the `for` loop executes about 340 machine instructions (x86) for the `orders` relation in executing the following query.

```
select o_comment from orders;
```

To execute the specialized code, we simply insert a function call to the `GetColumnsToLongs()` function to replace the `for` loop. The specialized code has only 146 instructions, for a reduction of approximately 190 instructions.

To determine the actual performance benefit, we studied the above query in detail. This query requests a sequential scan over the `orders` relation, which has 1.5M tuples (with the scale factor set to one for the TPC-H dataset). Given that the specialized code saves 190 instructions and the code is invoked 1.5M times (once per tuple), the total number of instructions is expected to decrease by 285M.

We utilized `callgrind` [11] to collect the execution profiles. The summary data produced by `callgrind` states the total number of executed instructions, the number of instructions for each function, and other runtime information. We first focus on the counts for the executed instructions.

We profiled the execution of this query with both a stock PostgreSQL and one with the shorter code replacing the `for` loop. (We elaborate in detail how such code is managed within the DBMS in Section IV.) The total number of executed instructions of the stock PostgreSQL was 3.447B, which implies that this micro-specialization will produce an (estimated) reduction of about 8.3%. The total number of instructions actually executed by the specialized PostgreSQL is 3.153B, a (measured) reduction of 8.5%, consistent with our earlier estimate. We then measured the total running time of the query on the stock PostgreSQL and the specialized version, at 734 milliseconds and 680 milliseconds, respectively. The improvement in running time (7.4%) is consistent with the profile analysis.

By specializing a single routine, the generic `slot_deform_tuple()` function, on just a few variables, we were able to achieve a 7.4% running time improvement on a simple query. This improvement suggests the feasibility and benefits of applying micro-specialization aggressively.

III. Approach

Each micro-specialization identifies one or more variables whose value will be constant within the query evaluation loop. It then replaces a function or small stretch of code with multiple copies, each particular to a single value of each of those variables. In the example given above, the variables concerned the relation being scanned. Hence, we need a specialized version of `GetColumnsToLongs()` for each relation.

We first introduce terminology for the specifics of our approach.

- The specialized code, in this case associated with a particular relation is termed a *bee*.
- A bee can have multiple *bee routines*, each produced by a particular micro-specialization at a certain place in the DBMS source code on one or more variables that have been identified as being invariant across the query evaluation loop.

In the example given above, micro-specialization is applied on values (attribute length, etc.) that are constant for each relation, and so a bee routine results. We term this particular bee routine `GCL`, as shorthand for the specialized `GetColumnsToLongs()` routine. There will be a unique *relation bee* for every relation defined in a database.

We specialized another PostgreSQL function named `heap_fill_tuple` that constructs a tuple to be stored from an long integer array, resulting in a separate bee routine namely `SetColumnsFromLongs()` (`SCL`) for each relation. So each relation bee now has two bee routines.

This general approach raises two central questions: *where* can micro-specialization be applied and *when* during the timeline from relation-schema definition to query evaluation can micro-specialization be done?

A. Where to Apply Micro-Specialization?

We present a taxonomy of approaches to micro-specialization in Figure 1, based on two types of "variables" in a DBMS where micro-specialization can be applied to: stored data and internal data structures. (This taxonomy also includes the other, coarser-grained specializations discussed in passing at the beginning of the paper.)

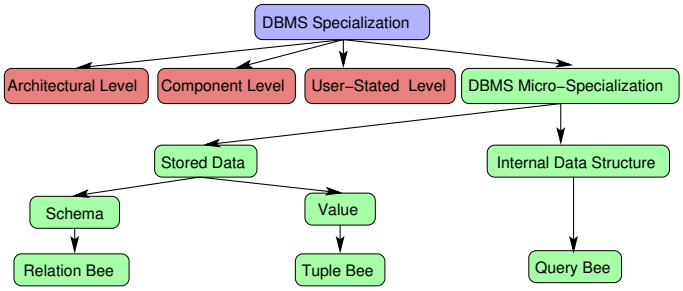

Fig. 1. The Taxonomy of *Where* to Apply Micro-Specialization in DBMSes

We previously discussed two bee routines within relation bees. These specialize code based on aspects of individual relations; hence, the specialization is that of the relational *schema*. In this particular case, we specialize on each attribute's length, offset, alignment, and the presence of nullable attributes, as well as on the number of attributes in the relation.

We can extend the application of micro-specialization down to an individual tuple by introducing *tuple bees*, in which specialization focuses on the values of particular attributes within a tuple. Consider an attribute with a few distinct values, such as "gender." When the value extraction routine requests the value of this attribute, instead of computing the length, offset, and alignment of the attribute, a single assignment such as `values[x] = 'M';` can properly fulfill the value extraction of this attribute. This occurs within a tuple bee associated with that tuple; we do so by including in such tuples a short index identifying the appropriate tuple bee, termed a *beeID*. So we might just have two tuple bees, one for each gender, or we might also specialize on other attributes, as long as there aren't too many tuple bees generated, so that a small number of tuple bees are generated for all the tuples in the relation.

The last type of bee specializes on internal data structure issued during query evaluation, for which some of the values in the data structure are constant during the evaluation loop of a query. For example, a query that involves predicates will utilize a `FuncExprState` data structure (a C `struct`) to encode the predicate. For the predicate `age <= 45`, this predicate data structure contains the ID of attribute `age`, the `<=` operator, and the constant 45. We can thus apply specialization on these variables once we know the predicate from the query. The bees resulting from specializing such query-related data structures are thus termed *query bees*.

This taxonomy characterizes three different kinds of bees, depending on the kind of variable specialized on to create the bee. By identifying values used by oft-executed code within the query evaluation loop, many bee routines can be created. Each bee routine will independently speed up a subset of queries.

B. When Can Micro-Specialization be Applied?

Figure 2 depicts when individual bees of each kind are created. Relation bees are created at relation schema definition time, one for each newly-created relation. Individual query bees are created during query plan generation. Once we have a query plan, we know the particulars of the various data structures used in query evaluation, and so can generate the highly-specific code that uses these structures. Tuple bees are created during the evaluation of tuple insertions and updates, deep within the query evaluation loop.

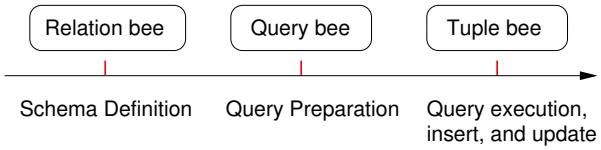

Fig. 2. *When* to Create Various Kinds of Bees

Each bee routine is the result of specializing on a type of variable, as emphasized in Figure 1. Note though that a variable available to an earlier specialization is thus also available to a later specialization. So for example a micro-specialization on an attribute's offset in a relation bee can both be utilized in a query bee or tuple bee. Hence, as we travel to the right along the timeline from schema definition to query execution, the number of variables available for micro-specialization accumulate, making those later bees highly efficient.

Where bee creation resides along the timeline affects how efficient bee creation must be. Note however that we are not discussing the design of a bee routine. As we will see in the next section, the code to create the individual bees and to invoke bee routines is manually inserted into the DBMS before it is compiled. Here we are focusing on creation of individual bees, each containing specialized code resulting from knowing the exact values of the variable(s) evincing the specialization.

For relation bees, bee creation overhead is not critical. Hence, when creating a relation bee, we can invoke *gcc* to compile the source code, as illustrated in Listing 2, to produce the executable object code for the bee.

Because ad hoc queries need to be fast, the overhead of generating query bees needs to be minimized. Recall that query bee may contain the join and predicate evaluation routines. In the case of join, all possible combinations of the join routines, such as different types of joins (left, semi, anti, etc.) can be enumerated and compiled ahead of time. At query preparation time, the associated join bee routine is assembled by selecting one of the pre-compiled join evaluation routines in the executable form.

Delving down into the details, there are two ways that specialization affects query bee code. Some specializations,

such as on the join type for a join evaluation bee routine in a query bee, affect the object code, thus resulting in multiple versions of a bee routine. Other specializations, such as the attribute ID for both the join and predicate bee routines, only affect constants in the bee routine. For the latter, the bee is effectively cloned each time, with different values substituted for the latter values, as discussed in Section V. Only the unique object code combinations need be generated ahead of time.

Finally, tuple bee generation needs to be extremely fast because the generation occurs during query execution. As discussed earlier, a tuple bee may contain the value(s) for particular attribute(s). As just mentioned, the bee can be cloned, with the particular value(s) then substituted. Creating tuple bees is thus very efficient, as we will demonstrate when we evaluate the performance in detail in Section VI.

It may seem that by creating individual bees, additional code is being added to the DBMS. In fact, the introduced code replaces the original code. Moreover, at run time a significant amount of instructions can be reduced by the specialized code, as illustrated in the case study.

IV. A Bee Architecture

We propose a separate *Generic Bee Module* which performs many of the underlying management tasks associated with creating bees and executing bee routines on behalf of the DBMS. The bee module provides an API to the DBMS, thereby making it easy for DBMSes to utilize micro-specialization while minimizing modifications to the existing DBMS.

Figure 3 depicts a conventional DBMS architecture [12]. The lightly-shaded boxes, including a repository (Templates), comprise the components of the bee module. This module provides the bee configuration, bee generation, and bee invocation functionality in a largely DBMS-independent fashion.

The shaded boxes represent the code added to the DBMS to invoke methods provided by the bee module API, as well as the other changes required to existing code within DBMS. To fully support all the bee types in the taxonomy of Figure 1, three existing DBMS components (the DDL Compiler, the Runtime Database Processor, and the Stored Data Manager), two repositories (the System Catalog/Data Dictionary and the Stored Database), and the schema (the DDL Statements) need to be augmented with added code (depicted with darker boxes). The thick lines denote calls to a component of the bee module and the dotted lines depict either storage of or access to schema information.

While the bee module comprises all of the functionality needed for incorporating bees into a DBMS (admittedly a good number of boxes in the figure), it is still quite manageable, as we'll see in the next section. One (continuing) challenge has been how to effectively partition the generic code from the DBMS-specific code additions, while ensuring that each bee routine provide added efficiency.

The developer performing micro-specialization, i.e., replacing some generic DBMS code with calls to specialized bee routines, must decide what bee routines are to be provided and effect the execution of bee routines in the context of query evaluation. The changes required to the architecture of a conventional DBMS to accommodate bees can correspondingly be classified into two groups, termed the *Bee Configuration Group* and the *Query Evaluation Group*, respectively. We now examine the components within each group.

A. The Bee Configuration Group

Each bee routine represents existing DBMS code that has been specialized based on invariant values, replaced with a call to that bee routine. The specific source code for that routine, for a particular bee, is generated by this group of components.

To generate an individual bee, the bee routines need to be specified by the developer performing micro-specialization. That developer uses whatever tools (e.g., code inspection, profiles) that are helpful. This manual task is outside of the scope of this paper; our only comment is that there seems to be a plethora of opportunities for such specialization within PostgreSQL. Each routine is assembled by the developer into a set of code *snippets*. For instance, to assemble the `GCL` bee routine discussed in the case study, the code snippets corresponding to each attribute are selected and grouped as the C source code shown in Listing 2. We term such resulting source code snippets, *templates*, in that they are not yet executable. Concerning a query bee that involves joins, we mentioned in the previous section that to avoid invoking *gcc* during query evaluation, we compose a code template with the join type and a few other variables specified as global variable, which can be optimized away when invoking *gcc* with these variables specified using the *-D* option.

As discussed in Section III-A, many tuples can share the same tuple bee. Moreover, all the tuples can in fact share the same relation bee routines such as `GCL`. The difference among various tuple bees is the data values. We create a clustered storage for all the distinct data values. We term this storage area the *data section*. To access the data section so that the tuple bees can respond with proper values to queries for particular tuples, the relation bee routines are modified to have "holes" for the specialized attributes, as shown in Listing 2, lines 7, 19, and 26. The `bee_id` argument is used to identify which data section a tuple bee is associated with. Therefore, it is necessary to store a bee ID along with each tuple. The magic numbers 1000, 1001, and 1002 shown in Listing 2 are used as identifying placeholders such that the correct data section addresses can be instantiated for a particular tuple bee.

Annotations are used in the creation of tuple bees, to specify which attributes, such as "gender," have small cardinalities. Annotations can be specified explicitly by the DBA or can be inferred (such as from SQL domains), a topic beyond the scope of this paper. The remaining component of the bee configuration group is bee reconstruction, triggered by changes in the schema of a relation.

B. The Query Evaluation Group

This collection of eight components all perform critical tasks to ensure that bees are properly managed by the bee module, coupled with actions within the DBMS itself.

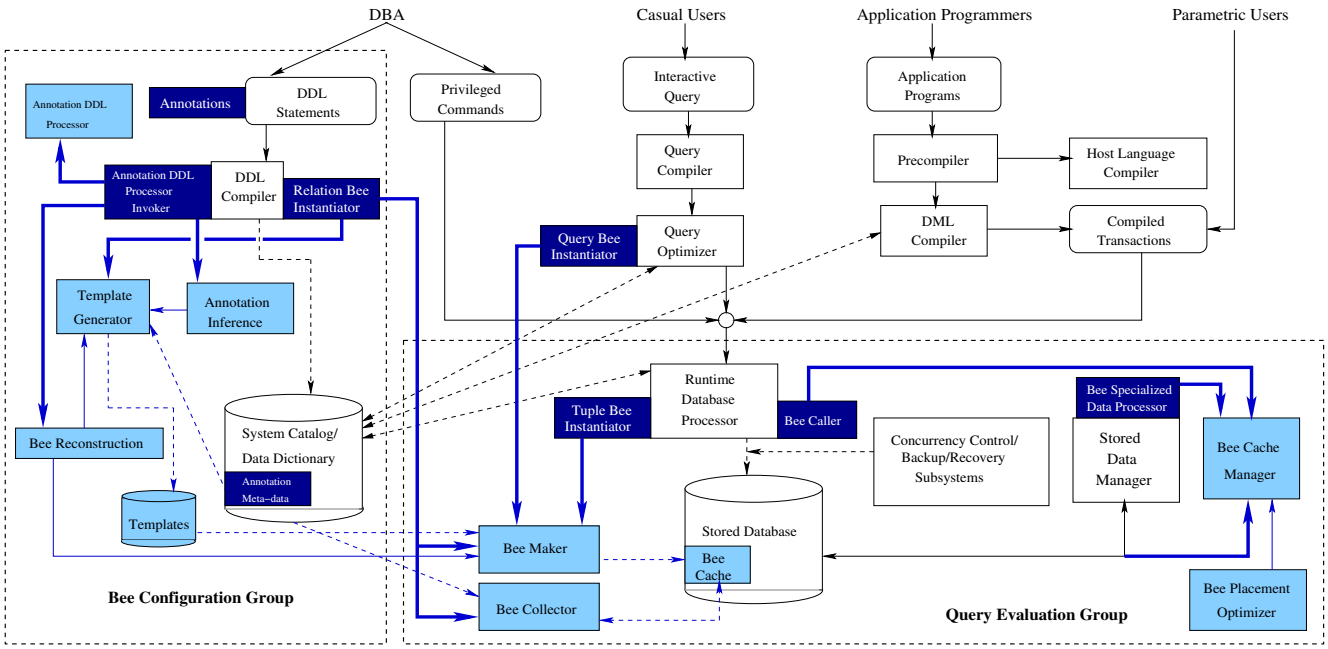

Fig. 3. Architecture of a Bee-Enabled DBMS

First, after the code templates for a particular bee are composed, the *Bee Maker* is invoked to compile the source code or replace placeholders in the object code. For relation bees, the bee maker is invoked at schema definition time. The bee maker relies on *gcc* to compile the template source code for bees. The resulting object file, namely *Executable and Linkable Format* (*ELF*) [13] contains a collection of information including headers, tables, variables, and function bodies, which are all useful when the ELF file is linked and loaded into memory for execution, in its conventional usage. However, to assemble a bee, only the function bodies corresponding to the bee routines are needed. So the bee maker extracts the function bodies and uses them to create the individual bee. For query bees, the bee maker is invoked from the query optimizer; for tuple bees, from the runtime database processor, specifically, the code that performs the tuple insertion or update. To avoid source code compilation during query preparation and evaluation when creating query and tuple bees, the template source code is pre-compiled. The resulting binary code is used to instantiate each executable bee routine. For query bees, instead of extracting the attribute ID and constant value in the comparison of a predicate with a general expression parsing routine, these two numbers can be inserted directly into the executable code, resulting in fewer instructions that reference these values as compared to the generic routine.

The resulting object code is stored in the *Bee Cache*, which is a repository of all the bees in executable form. The bee cache is written to disk along with the relations and database metadata.

The *Bee Cache Manager* component manages the bees when they are in main memory. When the bee templates are compiled into object code, the bees are formed and flushed to the on-disk bee cache. When the DBMS server starts, all the bees (or perhaps only the ones that are needed) are loaded into main memory so that bees can be directly invoked. Currently the bee cache is not guaranteed to survive across power failures or disk crashes, though a stable bee cache could be realized through the *Undo/Redo* logic associated with the log.

When a query is evaluated, the *Bee Caller* acquires the proper arguments. As an example, the `GCL` routine requires a pointer to the tuple (the `data` argument). The bee caller passes the needed arguments to the bee cache manager, and the bee cache manager invokes the proper bee routine with these arguments. The bees are placed at designated locations in memory such that the cache miss caused by executing the bees is minimized. Given that DBMSes at query evaluation involve a significant amount of instructions to be executed with many iterations, to effectively prevent bees from increasing further cache misses, bees are placed at memory locations that will not overlap with existing DBMS code in instruction cache. The placements are computed by the *Bee Placement Optimizer*. Note that we observed that the level-1 instruction cache miss rate is just around 0.3% across most TPC-H queries, therefore, the run time improvement produced by a proper placement of bees is trivial. However, we consider this component to be important in that it ensures an code locality optimized environment where more bees can be introduced with minimized impact on instruction cache.

Finally, the *Bee Collector* garbage collects dead bees (e.g., those not used anymore due to relation deletion), from those in both the bee cache manager (those bees in main memory) and in the bee cache on disk.

C. The Generic Bee Module

We have implemented initial versions of all of the components shown in Figure 3 to realize a fully-elaborated bee module, consisting of six thousand source lines of code (SLOC). This module is rather complex, as it has to deal with constructing bee routines from source code and from object code, it has to insert specific values into data sections within the object code, it has to place the object code so that that code occupies the desired cache lines, and it has to carry out all of these tasks efficiently.

V. Bee-Enabled PostgreSQL

The generic bee module provides functionality for creating and invoking relation, query, and tuple bees during query evaluation.

To evaluate the merits of micro-specialization generally as well as the architecture presented in the last section, we also implemented all three types of bees, some with several bee routines, in PostgreSQL. To do so, we first identified code sequences that simultaneously (i) appeared in the query evaluation loop, (ii) constituted a significant portion of the runtime of query evaluation, (iii) referenced variables whose value was invariant across the query evaluation loop, and (iv) could benefit significantly from micro-specialization, by removing branches and accesses on those variables.

We discussed in Section III two relation bee routines: the `GetColumnsToLongs()` function (the `GCL` bee routine) and the `SetColumnsFromLongs()` function (the `SCL` bee routine). The first routine is further specialized of specific attribute values, by placing the attribute value in a tuple bee.

An initial analysis identified 70 data structures in PostgreSQL that could be targeted for micro-specialization to create query bee routines. While not all are involved in every query, each is used by a subset of the queries, and some queries would use several such bees.

We created two query bee routines. One micro-specialized the predicate evaluation routine on the `FuncExprState` data structure, to realize an *evaluate predicate*, or `EVP`, query bee routine. As mentioned in the previous section, to avoid source code compilation during query preparation, the template source code for `EVP` is pre-compiled and specific values inserted into the object code during bee creation.

A second query bee routine micro-specializes the `JoinState` data structure, to realize the *evaluate join*, or `EVJ`, query bee routine. A relation join will utilize this data structure, containing the join comparison operator, the attribute IDs of the inner and outer tuples, and the type of join (e.g., anti-join, semi-join, left-join). These variables are all constant during the evaluation of a query. The values of these variables are inserted during bee creation.

For each of the relation, query, and tuple bees, we manually constructed the C or object code templates and inserted code in the DBMS to call the bee maker. We also inserted code to call the bee, thus replacing the generic code with highly-specialized code. We added the darkly-shaded boxes to the DBMS for memory allocation, resource reclamation, pointer assignments, and argument passing. The changes to PostgreSQL, which comprises 380K SLOC, was only about 600 SLOC for these components in concert.

We applied micro-specializations in parallel with the development of the generic bee module. Each new kind of bee requires extensions to the bee module. In the last few months of development, the bee module has settled down. The challenge remains in choosing the DBMS code to be micro-specialized and in creating the templates to be used during query evaluation by the bee maker.

VI. Empirical Evaluation

Micro-specialization replaces generic code containing many branches with highly customized code that relies on identified values being invariant in the query evaluation loop. The result is fewer instructions executed on each bee invocation, which when summed across the often millions of times around the loop can result in significant increase in performance.

We have investigated the performance impact of micro-specialization in many contexts: simple select queries such as discussed in the case study, OLAP-style queries and high-throughput bulk-loading in the TPC-H benchmark, and OLTP-style queries and modifications in the TPC-C benchmark.

To generate the dataset in TPC-H, we utilized the *DBGEN* toolkit [27]. The scale factor for data generation was set to one, resulting in the data of size 1GB. For TPC-C, we used the *BenchmarkSQL-2.3.2* [17] toolkit. The *number of warehouses* parameter was set to 10 when the initial dataset was created. Consequently, a total of 100 *terminals* were used (10 per warehouse, as specified in TPC-C's documentation) to simulate the workload. We also added DDL clauses to identify the handful of low-cardinality attributes the TPC-H relations. Other than specifying the scale factor and number of warehouses, we made no changes to other parameters used in the TPC-C and TPC-H toolkits for dataset preparation.

All the experiments were performed on a machine with a 2.8GHz Intel *i*7 860 CPU, which contains four cores. Each core has a 64KB Level-1 (*L1*) cache, which consists of a 32KB instruction (*I1*) and a 32KB data cache. The CPU is also configured with a 256K unified level-2 (*L2*) cache. Our prototype implementation used PostgreSQL version 8.4.2, compiled using *gcc* version 4.4.3 with the default build parameters (where the optimization level, in particular, is *–O2*).

A. The TPC-H Benchmark

We start with the TPC-H benchmark to compare the performance of the bee-enabled PostgreSQL with the stock DBMS. The TPC-H benchmark creates a database resembling an industrial data warehouse. The queries used in the benchmark are complex analytic queries. Such a workload, featured with intensive joins, predicate evaluations, and aggregations, involves large amount of disk I/O and catalog lookup. In particular, we examine the importance of applying bee placement optimization, which provides an improved cache-miss reduction framework, thus allowing bees to be more efficiently

executed. In studying bulk-loading, we quantify the running time improvement in populating the same relations.

All 22 queries specified in TPC-H were evaluated in both the stock and bee-enabled PostgreSQL. The running time was measured as wall-clock time, under a warm-cache scenario. We first focus on the warm-cache scenario to study the CPU performance: keeping the data in memory effectively eliminated the disk I/O requests.

We ran each query twelve times. The highest and lowest measurements were considered outliers and were therefore dropped. The running time measurement for each query was taken as the average of the remaining ten runs.

To ensure the validity and repeatability of the results, we tried to ensure that in evaluating these 22 queries, both the stock and the bee-enabled PostgreSQL were in fact using the same query plans. It was difficult to ensure that the two DBMSes would always choose the same plan, especially as the underlying relations had different characteristics under the two DBMSes through micro-specialization, e.g., the relation size, tuple size, and number of pages occupied by a relation. However, by setting the "default_statistics_target" parameter in the `postgresql.conf` file to 1000 (100 by default), we were successful in ensuring 21 of the queries were using the same plan across the two DBMSes. The only query with different plans was *query21*.

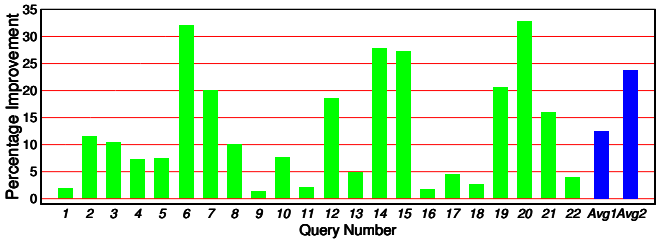

Fig. 4. TPC-H Run Time Improvement (Warm Cache)

Figure 4 presents the percentage performance improvements for each of the 22 queries with a warm cache, shown as the green (lightly shaded) bars. We include two summary measurements, termed *Avg1* and *Avg2*, shown as the blue (more darkly shaded) bars. *Avg1* is computed by averaging the percentage improvement over the 22 queries, such that each queries is weighted equally. *Avg2* is computed by comparing the sum of all the query evaluation time. Given that *query17* and *query20* took much longer to finish, about one hour and two hours, respectively, whereas the rest took from one to 23 seconds, *Avg2* was highly biased towards these two queries. The range of the improvements is from 1.4% to 32.8%, with *Avg1* and *Avg2* being 12.4% and 23.7%, respectively. In this experiment, we enabled tuple bees, relation bees, and query bees, involving the `GCL`, `EVP`, and `EVJ` bee routines. As shown by this figure, both *Avg1* and *Avg2* are significant, indicating that the performance improvement achieved in the bee-enabled PostgreSQL are generally applicable.

To ascertain the I/O improvement achieved by tuple bees, we then examined the run time of the 22 queries with a cold

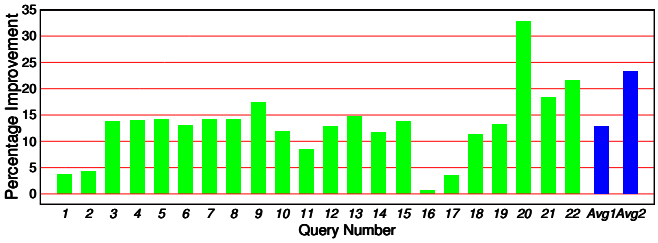

Fig. 5. TPC-H Run Time Improvement (Cold Cache)

cache, where the disk I/O time becomes a major component of the overall run time. Figure 5 presents the run time improvement with a cold cache. The improvement ranges from 0.6% to 32.8%, with *Avg1* being 12.9% and *Avg2* 22.3%. A significant difference between this figure and Figure 4 is that the performance of *q9* is significantly improved with a cold cache. The reason is that *q9* has six relation scans. Tuple bees are enabled for the `lineitem`, `orders`, `part`, and `nation` relations. Therefore, scanning these relations, in particular the first two benefits significantly from attribute-value specialization and thus the near 17.4% improvement is achieved with a cold cache.

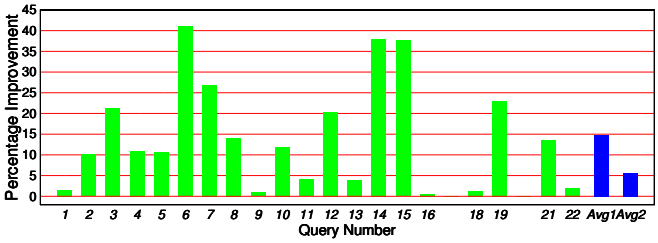

Fig. 6. Improvements in No. of Instructions Executed

Figure 6 plots the improvements in the number of instructions executed for each query. The reductions in dynamic instruction count (shown as the lightly-shaded green bars in Figure 6) range from 0.5% to 41%, with *Avg1* and *Avg2* of 14.7% and 5.7%, respectively. Note that when profiling with `callgrind`, program execution usually takes around two hundred times longer to finish. We were thus not able to collect the profile data for *q17* and *q20*, Therefore, we omitted the profile related results for these two queries. This plot indicates that the running time improvement is highly correlated with the reduction of instructions executed, further emphasizing that the benefit of micro-specialization stems from the reduced instruction executions.

Performance improvement for each query is accomplished by all the bees that are invoked. Recall that in Section II, just the `GCL` routine of a relation bee achieved 7.4% improvement. A fundamental question is that how much improvement can be further achieved by adding more bees? More importantly, would many bees adversely impact each other?

We examine the effect of enabling various bee routines. We summarize the results in Figure 7. As shown by this figure, the average improvement with just the `GCL` routine is 7.6% for

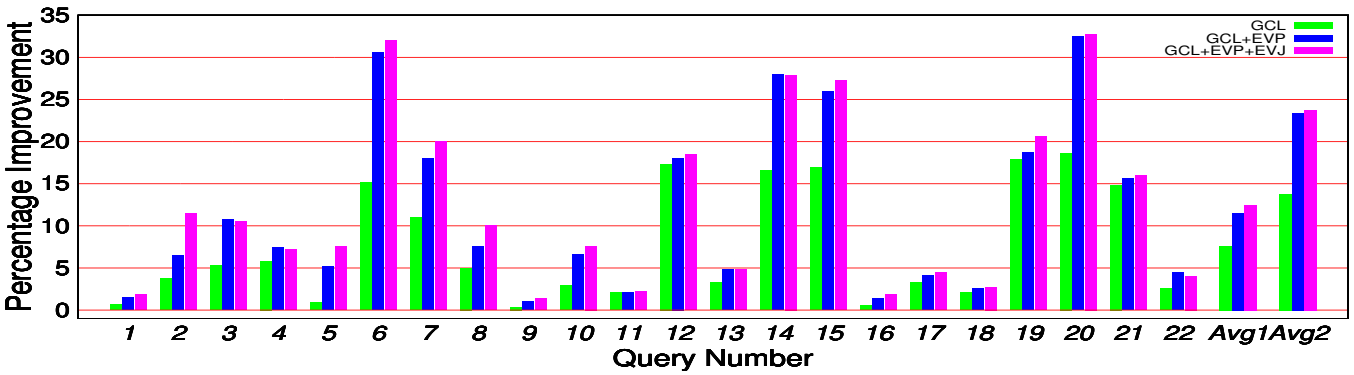

Fig. 7. TPC-H Run Time Improvement with Various Bee Routines Enabled (Warm Cache)

Avg1 and 13.7% for *Avg2*. By enabling the `EVP` routine, the average improvement reaches up to 11.5% (*Avg1*) and 23.4% (*Avg2*). Among all the queries, *q6* shows the most significant improvement, from 15.1% to 30.6%, by enabling `EVP` on top of `GCL`. This is because *q6* contains complicated predicates whereas the the query scans just one relation. Finally, we enable all three bee routines. Although the overall improvement is slightly increased, we found that a few queries, such as *q2* and *q5* were improvement significantly. Not surprisingly, both queries have complicated join condition evaluations. A key observation is that by adding more bee routines, the improvement achieved by the already enabled routines is not compromised. (The small decrease in running time when all three bee routines are enabled for queries such as *q4* and *q22* is due to measurement error.) The implication is that the micro-specialization approach can be applied over and over again. The more places micro-specialization is applied, the better efficiency that a DBMS can achieve. We term this property of incremental performance achievement *bee additivity*.

Most performance optimizations in DBMSes benefit a class of queries or modifications but slow down others. For example, B^+-tree indexes can make joins more efficient but slow down updates. Bee routines have the nice property that they are only beneficial (with two caveats to be mentioned shortly). The reason is that if a bee routine is not used by a query, that query's performance will not be affected either way. On the other hand if the bee routine is used by the query, especially given that the bee routine executes in the query evaluation loop, that query's performance could be improved considerably. N ote that both Figure 4 and Figure 7 show difference among the performance improvements. For instance, *q1*, *q9*, *q16*, and *q18* all experience relatively lower improvements. The reason is that these queries all have complex aggregation computation as well as sub-query evaluation that have not yet been micro-specialized with our implementation. These queries with low improvement point to aggregation and perhaps sub-query evaluation as other opportunities for applying micro-specialization.

B. Bulk-Loading

A concern is that tuple bee creation during modifications, such as populating a relation, may be expensive, in that the specialized attribute values from a newly inserted tuple need to be examined to determine if a new tuple bee is needed. Moreover, when a new tuple bee is created, new memory space needs to be allocated to store this bee. To ascertain the possible performance hit of this second caveat, we performed bulk-loading on all the relations in the TPC-H benchmark.

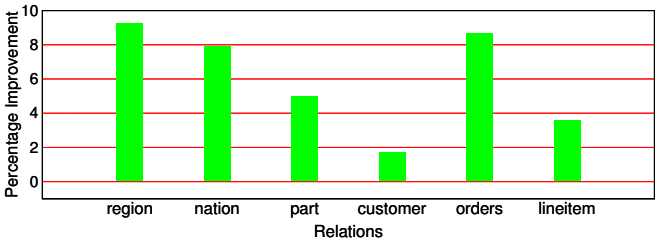

Fig. 8. Bulk-Loading Run Time Performance

We compared the bulk-loading performance of the bee-enabled PostgreSQL with the stock version. Since no query evaluation is performed in bulk-loading, only the `SCL` bee routine is involved. Figure 8 presents for each relation the loading time speed-up. In the TPC-H benchmark, the `region` and `nation` relations each occupies only two disk pages, which makes the performance impact of loading the two relations not measurable. Therefore, we created for each relation a data file that contains 1M rows. The performance of loading these two relations reported in Figure 8 is based on populating these two relations each with 1M rows. The rest of the measurements comply with the original schema and data.

The bulk-load performance improvement presented in Figure 8 suggests that the overhead of creating tuple bees during bulk-loading is in fact compensated by the benefit of micro-specialization, such that the overall bulk-load performance is improved. To understand where the performance improvement comes from, we studied the profile of bulk-loading the the `orders` relation. In bulk-loading this relation, the stock PostgreSQL executed 148B instructions. Whereas the bee-enabled PostgreSQL executed 146B instructions. In the stock execution, the `heap_fill_tuple` function that constructs a physical tuple executed 4.6B instructions. The bee-enabled PostgreSQL utilizes the `SCL` bee routine to replace

this generic function. The SCL routine executed 2.4B instructions. However, the saving does not fully explain the 8.3% running time improvement shown in Figure 8. The remainder of the improvement comes from attribute-value specialization. By utilizing tuple bees, some specialized attribute (distinct) values are stored within the bees rather than in the relations, as discussed in Section IV-A, achieving significant disk space and thus I/O savings, as suggested by Figure 8.

When distinct attribute values need to be store in a tuple bee, the slab-allocation technique is employed to pre-allocate the necessary memory, therefore avoiding expensive small and frequent memory allocation during tuple bee creation. To determine whether a new tuple bee is needed, we check the few (maximally 256) possible values with *memcmp*. Figure 8 indicates that this step is efficient.

In summary, bee creation does not adversely impact the performance of DBMS operations; rather, the performance is improved due to the benefit of even a single bee routine.

C. The TPC-C Benchmark

The TPC-C benchmark focuses on throughput. This benchmark involves five types of transactions executing in parallel. The throughput is measured as the number of *New-Order* transactions processed per minute (tpmC). The other four types of transactions produce a mix of random queries and modifications, which altogether intensively invoke the bee routines.

Our experiments compared the bee-enabled PostgreSQL with the stock DBMS. Each DBMS was run for one hour, to reduce the variance introduced by the experimental system as well as the DBMS, e.g., the auto vacuum processes.

Performing modifications with micro-specialization was actually faster: the former completed 1898 transactions per minute while the stock DBMS could execute 1760 transactions per minute, an improvement of 7.3%.

We moved beyond this tpmC metric of the TPC-C benchmark and studied the throughput with different transaction settings. We focused on two more quite different scenarios. Of the five defined transaction types, three of them, *New-Order*, *Payment*, and *Delivery* include both queries and modifications; *Order-Status* and *Stock-Level* on the other hand only contain queries. For both scenarios, the weight of the *New-Order* transactions was kept at 45%. The default setting resembles a modification-heavy scenario in that the weight of the *Payment* transaction is 43%. Regarding our newly defined scenarios, the first consists of 27% order-status and 28% stock-level transactions (that is, only queries). The second scenario has an equal mix of both modifications and queries. The weight of the *Payment* and the *Delivery* transactions is 27% whereas the other two types of transactions are weighted 28% in concert.

For the first scenario, that of only queries, the bee-enabled PostgreSQL and the stock DBMS handled 3699 and 3135 transactions per minute, respectively, for an improvement of 18%. Concerning the second scenario, with modifications and queries equally weighted, the bee-enabled PostgreSQL achieved 2220 transactions and the stock version finished 1998. The improvement is 11.1%.

The profile results suggested that both modifications and queries rely on the slot_deform_tuple function to extract tuple values. Since this function is micro-specialized with the GCL routine, significant performance improvement is achieved for various scenarios in the TPC-C benchmark. Moreover, since the queries in this workload involves predicates, the EVP routine has also contributed to the improved throughput, particularly to the query-heavy scenarios.

VII. Related Work

DBMS specialization is a common and effective approach to increasing performance of DBMSes. These specialization approaches can be applied independently over a wide spectrum. In the following, we give a necessarily incomplete sampling of such specializations, to show what they have in common with micro-specialization and also to emphasize how micro-specialization differs from these other approaches.

a) Architectural Specialization: At the coarsest level, *architectural specialization* customizes the entire architecture to a particular, sometimes quite narrow application domain. An example is the column-oriented storage model utilized by several novel DBMSes, such as ColumnDB [1], C-store [23], and MonetDB [7]. Column-oriented stores are very effective in OLAP applications, by reducing the I/O overhead by fetching only the necessary columns from disk during query evaluation. In addition, MonetDB utilizes a vectorized approach that models each column as a vector [8]. MonetDB uses a BAT representation, with each table having just two columns, enabling the join operators to be specialized to particular types ahead of time. Micro-specialization, because it focuses on short code sequences within the DBMS, can be applied at many places, independently of architectural specialization.

To address OLTP workloads, VoltDB [30], which is based on *H-store* [24], operates on a distributed and shared-nothing architecture. By adopting a single-threaded in-memory execution model, VoltDB can eliminate the overhead of locking, buffering, logging, and latching mechanisms. Although many components can be eliminated from a DBMS such as VoltDB, micro-specialization is still applicable to such DBMSes. Take a query predicate as an example. VoltDB still utilizes data structures, of which values can be invariant, in evaluating such predicates. As suggested by Figure 8, micro-specialization improves the performance of predicate evaluation significantly by simply specializing on the predicate operands and operator type, such that the complexity of the generic code that references these variables and data structures are greatly reduced.

Real-time [3] and stream DBMSes [2] are other examples of architectural specialization, in which the entire database architecture is drastically modified to fit a restricted subset of applications. Such an approach can deliver a significant performance improvement. Micro-specialization could further improve the performance of such DBMSes.

Krikellas et al. employed an approach to produce specialized code to replace the entire original generic query eval-

uation routines implemented in conventional DBMSes [16]. The proposed method uses code templates to form the specialized code for processing specific queries. The code is then compiled and executed to evaluate the queries. The scope of the code replacement is vast: the entire query evaluation code base, often tens or hundreds of thousands of lines, must be moved into templates that are then stitched together. These can be also characterized as architectural specializations, reflecting their impact on the structure of the DBMS.

Sompolski et al, Rao et al, and Neumann [19, 20, 22] exploit similar mechanisms in compiling queries during their execution. The MAL language [7] provided by MonetDB also achieves a similar goal of generating specialized code during query execution by tailoring the primitives utilized by query evaluation. These approaches are also much coarser-grained than the micro-specialization proposed here.

b) Component Specialization: A DBMS may also benefit from *component specialization*, where multiple versions of a single module of the DBMS are provided, each customized to a particular kind of data or query through the use of a new data structure or algorithm. Examples include new relational operators, new indexing methods, new isolation levels, added compression, and additional locking modes in concurrency control. Each addition improves the performance of a specific subset of the applications. Micro-specialization does not start with particular component. Instead, our approach identifies the invariants from the code of a DBMS first. Regardless of the origins of these invariants, which can be join, scan, or sort operators, all invariants can be specialized on by replacing the original generic code with a specialized version.

There has been exciting work in what is termed *architecture-conscious optimizations* [9], such as reducing data cache misses in DBMSes by re-organizing data page layout [4, 5] or by data partitioning [18, 21], blocking, as well as clustering [21]; reducing instruction cache misses by re-structuring the code execution paths as well as keeping instructions in cache for sharing [15, 31]; and minimizing cache stall latency with prefetching strategies [10]. These particular efforts can be classified generally as component specializations and thus are orthogonal to (finer-grained) micro-specialization, which is itself an architecture-conscious optimization that has as its goal to reduce instruction executions (and thus as a side effect both data-cache and instruction-cache misses).

c) User-Stated Specialization: A third type of DBMS specialization is *user-stated specializations* that require user involvement and generally employ SQL query language constructs such as triggers. The SQL code in triggers can be optimized by the user. Another example is user-defined functions, which can be implemented in various programming languages and invoked in SQL statements. UDFs are often much more efficient than pure SQL, and are more flexible, as they are expressed in a Turing-complete programming language. That said, the DBMS code to support triggers and UDF invocation can itself be micro-specialized for further improvement.

d) Characteristics of Micro-Specialization: Micro-specialization is applied at a finer granularity than any of the above specializations. Micro-specialization is applied to a short sequence of low-level query evaluation code. Hence, it is orthogonal to and independent of other coarser-grained specializations, enabling micro-specialization to be aggressively applied equally well to conventional DBMS architectures (e.g., PostgreSQL, IBM DB2, Oracle, and Microsoft SQLServer), to column-oriented stores such as MonetDB, ColumnDB, and C-store, to OLTP architectures such as VoltDB, and to real-time and stream DBMSes. And it can be applied to various modules arising from component specialization, and in conjunction with user-stated specializations, e.g., within the code sequences that implement triggers. Any code within a DBMS that is executed frequently and involves variables that are invariant over a single time around the inner per-tuple processing loop is a potential target for micro-specialization. Finally, micro-specialization instantiates bees at various points along the timeline, providing flexibility with regard to when such instantiation is best performed.

e) The Template Mechanism: Interestingly, implementations of C++ templates [25, 28] and Java generics [6] concern specialization of the methods of a parameterized class according to the provided parameters. The C++ template mechanism enables (as side effect) computation to be performed at compile time via techniques termed *metaprogramming* and *partial evaluation* [29]. Both techniques specialize programmer-stated templates at compile time by exploiting known parameter values and code structures, such that the generated code is optimized. Micro-specialization is different in that the specialization can be later, benefiting from run time information that is not available by static analysis for compile-time optimizations.

The template mechanism can potentially serve as a more organized approach to applying micro-specialization for bees instantiated at compile time. For instance, for relation bees, code snippets are extracted for various attribute types first. These snippets are then assembled by the generic bee module during bee instantiation, where a set of operations, including unrolling the attribute extraction loop, constant folding the `isnull` array, as well as choosing the specific code snippets to apply to each attribute type, are carried out, resulting in code shown in Listing 2. We expect that such operations could instead be performed by utilizing templates. For instance, functions that extract attributes with pass-by-value type of values can be defined as a function template. The length of such values can be specified as a *template parameter* to instantiate the function template. Other types of value extraction functions can be defined as templates similarly. When a relation bee is instantiated, the bee routine source code, which consists simply a list of function calls that instantiate specific types of functions, will be compiled. In this way, bee instantiation is performed by the compiler and the generic bee module is simplified.

VIII. Conclusion and Future Work

We have introduced a novel form of DBMS specialization, targeting small sequences of code, termed *micro-specialization*. This perspective utilizes the concept of *bees*, which are highly optimized code fragments obtained by dynamic code specialization based on variables whose values are invariant within the query evaluation loop. Bees contain *bee routines* that can be invoked by the DBMS; these replace code in conventional DBMS while performing the same operations more efficiently. The generality of DBMS is preserved by micro-specialization. Moreover, micro-specialization does not change the architecture of the DBMS nor does it add significant complexity to DBMS.

We have implemented the generic bee module and micro-specialized six bee routines across relation, query, and tuple bees integrated into the PostgreSQL DBMS. We have studied the performance of the resulting bee-enabled PostgreSQL, focusing on bee creation performance, CPU performance in complex analytic queries, and performance of random modifications. The bee-enabled PostgreSQL has achieved around 12% improvements over the stock version, simultaneously in I/O and CPU time, with the TPC-H analytic queries. For various scenarios of intensive TPC-C modifications, improvements of around 11% have been achieved as well, showing that micro-specialization is generally applicable regardless of specific types of workloads.

We plan to further investigate the many opportunities in performance improvement within other DBMS operations such as aggregation and indexing. Furthermore, since micro-specialization is orthogonal to other DBMS specialization approaches, we can apply this approach to other architectures, for instance, a column-oriented DBMS. Finally, we plan to develop a robust bee cache recovery component in the bee module to utilize Undo/Redo logs to ensure the stability of the bee architecture. While exploiting new specialization opportunities, we will further complete the bee module: incorporating additional bee components as well as refine the existing ones. We also plan to considerably enhance the bee development environment.

Acknowledgements

The work of Rui Zhang and Richard Snodgrass was supported in part by the National Science Foundation under grants IIS-0803229 and IIS-1016205, and of Saumya Debray under NSF grants CNS-1016058 and CNS-1115829. We also thank the reviewers for their valuable suggestions.

References

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 671–682, New York, NY, USA, 2006.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12:120–139, August 2003.

[3] R. K. Abbott and H. Garcia-Molina. Scheduling Real-time Transactions: A Performance Evaluation. *ACM Trans. Database Syst.*, 17:513–560, September 1992.

[4] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 169–180. Morgan Kaufmann, 2001.

[6] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. *PVLDB*, 2(2):1648–1653, 2009.

[8] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[9] D. Carmean, B. Falsafi, B. C. Kuszmaul, J. M. Patel, and K. A. Ross. Architecture-Conscious Databases: Sub-Optimization or the Next Big Leap? In *DaMoN*, 2005.

[10] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance through Prefetching. *Data Engineering, International Conference on*, 0:116, 2004.

[11] Valgrind Developers. Callgrind: A Call-Graph Generating Cache and Branch Prediction Profiler. http://valgrind.org/docs/manual/cl-manual.html (accessed October 27, 2010).

[12] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison Wesley Publishing Company, sixth edition, April 2010.

[13] Linux Foundation. ELF and ABI standards. http://refspecs.freestandards.org/elf/elf.pdf (accessed October 10, 2010).

[14] PostgresSQL Global Development Group. PostgresSQL. http://www.postgresql.org/ (accessed August 29, 2010).

[15] S. Harizopoulos and A. Ailamaki. Steps towards Cache-Resident Transaction Processing. In M. A. Nascimento, M. Tamer Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 660–671. Morgan Kaufmann, 2004.

[16] K. Krikellas, S. Viglas, and M. Cintra. Generating Code for Holistic Query Evaluation. In *ICDE*, pages 613–624, 2010.

[17] D. Lussier. BenchmarkSQL. http://sourceforge.net/projects/benchmarksql/ (accessed August 15, 2010).

[18] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[19] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB*, 4(9):539–550, 2011.

[20] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, page 23, 2006.

[21] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the VLDB*, pages 510–521. Morgan Kaufmann, 1994.

[22] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. Compilation in Query Execution. In *DaMoN*, pages 33–40, 2011.

[23] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-Oriented DBMS. In *Proceedings of the VLDB*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[24] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the VLDB*, pages 1150–1160. VLDB Endowment, 2007.

[25] B. Stroustrup. *The C++ programming language - special edition (3. ed.)*. Addison-Wesley, 2007.

[26] TPC. TPC Transaction Processing Performance Council - TPC-C. http://www.tpc.org/tpcc/ (accessed August 29, 2010).

[27] TPC. TPC Transaction Processing Performance Council - TPC-H. http://www.tpc.org/tpch/ (accessed August 29, 2010).

[28] D. Vandevoorde and N. M. Josuttis. *C++ Templates – the Complete Guide*. Addison-Wesley, 2003.

[29] T. L. Veldhuizen. C++ templates as partial evaluation. *CoRR*, cs.PL/9810010, 1998.

[30] VoltDB Inc. How VoltDB Works. http://voltdb.com/content/how-voltdb-works (accessed January 17, 2011).

[31] J. Zhou and K. A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. In *SIGMOD Conference*, pages 191–202, 2004.