Generating code for holistic query evaluation

Konstantinos Krikellas [1], Stratis D. Viglas [2], Marcelo Cintra [3]

School of Informatics, University of Edinburgh, UK

[1] k.krikellas@sms.ed.ac.uk
[2] sviglas@inf.ed.ac.uk
[3] mc@inf.ed.ac.uk

***Abstract*— We present the application of customized code generation to database query evaluation. The idea is to use a collection of highly efficient code templates and dynamically instantiate them to create query- and hardware-specific source code. The source code is compiled and dynamically linked to the database server for processing. Code generation diminishes the bloat of higher-level programming abstractions necessary for implementing generic, interpreted, SQL query engines. At the same time, the generated code is customized for the hardware it will run on. We term this approach *holistic query evaluation*. We present the design and development of a prototype system called HIQUE, the *Holistic Integrated Query Engine*, which incorporates our proposals. We undertake a detailed experimental study of the system's performance. The results show that HIQUE satisfies its design objectives, while its efficiency surpasses that of both well-established and currently-emerging query processing techniques.**

I. Introduction

This paper presents the application of customized code generation for the purpose of efficient database query processing. Our approach stems from template-based programming. The idea is to use code templates for the various query processing algorithms and then dynamically instantiate them and compose them in a single piece of source code that evaluates the query. Dynamic template instantiation removes the deficiencies of all high-level abstractions that are necessary for implementing generic query evaluators in current query engine designs. Moreover, since the code is dynamically generated, it can be customized to exploit the architectural characteristics of the hardware it will execute on. We term this approach *holistic query evaluation*, as the key premise is that one should take a holistic view of both the query being evaluated and the host hardware. The resulting performance advantage in main-memory execution is substantial; for instance, it reaches a factor of 167 over established database technology in TPC-H Query 1. The novelty we claim is that template-based code generation can be generalized to efficiently process any type of query without affecting orthogonal aspects of the database system.

Motivation. Traditionally, query processing algorithms have focused on minimizing disk I/O while their in-memory efficiency has been considered to be a secondary priority. For contemporary servers with large amounts of memory, it is conceivable for a large portion of the on-disk data –or even the entire database– to fit in main memory. In such cases, the difference in access latency between the processor's registers and main memory becomes the performance bottleneck [1]. To optimize such workloads, one needs to carefully "craft" the executed code so as to minimize the processor stall time during query execution.

Existing work has identified the data layout as the main bottleneck that prevents contemporary processor designs with multiple levels of cache memories from exploiting their full potential in database workloads. We argue that changing the storage layer is a radical departure from existing designs. We identify the biggest problem with the design of a query engine to be the compilation of SQL queries in operator plans and the generality of the common operator interface, namely the *iterator model*. The latter results in a poor utilization of CPU resources. Its abstract implementation and the frequent use of function calls inflate the number of instructions and memory accesses required for query evaluation. The use of generic code does not permit its customization according to the characteristics of both the executed queries and the hardware platform. SQL and query processing in main memory, however, exhibit a strong potential for exploiting just-in-time compilation. We take this idea to the extreme.

Code generation for query evaluation. Ideally, query processing code should optimally use the cache hierarchy and reduce the number of instructions needed for query evaluation. At the same time, one would want to keep the compositional aspects of the iterator model and not affect separate system modules. To that end, we introduce a novel query evaluation technique that we term *holistic query evaluation*. The idea is to inject a *source code* generation step in the traditional query evaluation process. The system should look at the entire query and optimize it holistically, by generating query- and hardware-specific source code, compiling it, and executing it.

Our approach has multiple benefits: (*a*) the number of function calls during query evaluation is minimized; (*b*) the generated code exhibits increased data locality, therefore making optimal use of cache-resident data; (*c*) code generation and compilation allow the use of compiler optimization techniques targeting each individual query, an extra optimization level on top of conventional query optimization; and (*d*) the generated code approaches the performance of hard-coded evaluation plans. The model is flexible and does not affect other orthogonal system aspects, such as storage management and concurrency control.

Using this framework, we have developed a prototype holistic query engine and compared its performance to both iterator-based solutions and existing database systems. The results (*a*) quantify the advantage of per-query code generation over generic query operator implementations, and (*b*) demonstrate a superiority of the holistic approach over both iterator-based and hardware-conscious systems in a subset of the TPC-H

978-1-4244-5446-4/10/$26.00 © 2010 IEEE

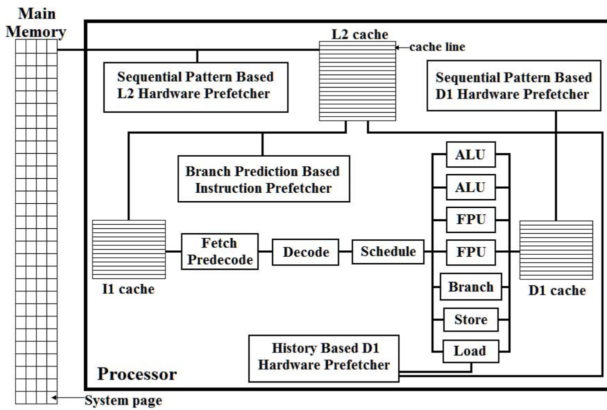


Fig. 1. The architecture of modern CPUs

benchmark, therefore proving its viability as an alternative query engine design.

The rest of this paper is organized as follows: in Section II we describe current processor and compiler technologies and conventional query engine design. Related work is presented in Section III. In Section IV we present the design of a system based on holistic evaluation. Our approach to code generation and our query evaluation algorithms are given in Section V, while in Section VI we experimentally evaluate the performance of the proposed model. Finally, we draw conclusions and identify future research directions in Section VII.

II. BACKGROUND

A. Hardware primer

Modern CPUs process multiple instructions simultaneously through pipelining and superscalar execution [14]. If the CPU supports out-of-order execution, instructions waiting for data transfer or other instructions to execute first yield for ready-to-execute instructions following in the pipeline. Out-of-order execution hides stalls, but offers limited advantages when the executed code includes consecutive memory requests and long data and control dependency chains, which are common in database workloads.

The substantial latency for transferring data from main memory to the processor's registers is countered with multiple levels of cache memory. The closer a level is to the processor, the smaller and faster it is to access it. Cached data is organized in fixed-length chunks, each termed a *cache line*: the data exchange unit with the main memory. Modern processors incorporate a very fast to access level-1 (L1) cache, divided in the Instruction (I1) cache and the Data (D1) cache; a larger but moderately fast to access level-2 (L2) cache; and, in some models, an even larger but much slower to access level-3 (L3) cache. Caches exploit both temporal locality (data tend to be repeatedly accessed within a short period) and spatial locality (contiguously allocated data tend to be accessed in unison). Non-blocking operation and superscalar execution allow for multiple pending memory operations, thus overlapping fetch latencies. Data-intensive workloads restrict this operation: the cache controller can only serve a limited number of concurrent requests and therefore becomes saturated. While caches can serve most of the processor's instruction and data requests, the remaining ones (cache misses) are expensive and can become a performance bottleneck.

To aid the caches, latest processors incorporate hardware prefetchers that identify the instructions and data likely to be accessed shortly and prefetch them into the appropriate cache level [10]. Current CPUs, as shown in Figure 1, employ multiple prefetching units tightly coupled with the cache hierarchy. The simplest ones are capable of detecting sequential patterns. The more advanced ones closely monitor the addresses touched by the processor to identify more complex access patterns by (*a*) keeping the history of accesses for a small number of the most frequently accessed addresses, and (*b*) tracking the distance (stride) between successive fetches.

To quantify the impact of hardware prefetching, we measured the data access latency for sequential and random access inside the memory hierarchy of a system using an Intel Core 2 Duo 6300 processor, clocked at 1.86GHz.[1] The results showed that, while all accesses to the D1-cache have a uniform cost of 3 CPU cycles, there is a significant difference when switching from sequential to random access in the L2-cache: the former takes 9 cycles and the latter 14 cycles. The gap grows when a data access cannot be served from caches, as sequential access in main memory costs 28 cycles, while random access costs 77 cycles or more.

B. Drawbacks of iterators

Most query engines are based on the iterator model [11]. This model provides an abstract interface used for streaming tuples across query operators, in the form of three functions: (*a*) `open()`, designating the start of information exchange and initialization of internal operator state, (*b*) `get_next()`, for propagating tuples between operators, and (*c*) `close()`, denoting the end of processing and allowing the operators to free up their resources. Query plans are then organized as pipelined binary trees of operators communicating through iterator calls.

Though generic, the iterator model exhibits a large number of function calls. For each in-flight tuple the system makes at least two calls: one for the caller to request it and one for the callee to propagate it. The number of function calls further grows as iterators need to be generic. Their functions may be virtual to be dynamically bound to the data types they process, which implies that all field accesses and comparisons may require a function call. Each function call updates the stack register and saves/restores the contents of the CPU registers to/from the stack. With tens of registers in current CPUs, frequent function calls may lead to significant overhead, as a substantial percentage of CPU time is spent without any actual contribution to result computation. Moreover, since a function call is a jump in the executed code, it forces a new instruction stream to be loaded in the pipeline, thus limiting superscalar execution.

In addition to stack interaction, there is also overhead at the data level. Each iterator maintains internal state; iterator

[1]The results were extracted using the RightMark Memory Analyzer [22].

calls require several memory operations for accessing and updating the iterator state, each call potentially triggering cache misses. Moreover, iterator state manipulation interferes with data stream accesses. Even if the data access pattern is sequential it will be frequently interrupted, thus reducing the efficiency of hardware prefetching. Note that the iterator interface does not control the data flow of pipelined operators, as each operator implementation is independent. Consequently, pipelined iterator calls may introduce cache contention and evict cache lines from each other's data-set, leading to cache thrashing.

C. Compiler optimization

Developers rely on the compiler to transform the code in ways that reduce processor stalls during execution. Since the compiler generates the executable code, it can optimize it for the target architecture and hardware platform. The code is transformed in ways that (*a*) keep the execution pipeline full of independent instructions, (*b*) distribute variables to registers in ways that encourage their reuse, and (*c*) group together accesses to the same data [16]. These optimizations result in increased parallelism, reduced memory accesses and maximized cache locality, thus limiting processor stalls.

The iterator model, however, prevents the compiler from applying such optimizations. Each iterator call triggers a chain of function calls that will eventually produce a single tuple. The compiler cannot factor this out and identify the (possibly iterative) access pattern over the input, as interprocedural analysis and optimizations are much more limited than intraprocedural ones. Moreover, conditions and jumps in the code due to function calls disrupt the instruction sequence and reduce the range of code the compiler can examine in unison for optimization opportunities. This is aggravated by certain parameters necessary for iterator instantiation (*e.g.*, predicate value types and offsets inside tuples) being only specified at run-time. These ambiguities refrain the compiler from applying a substantial part of its code optimization techniques on iterator implementations.

III. RELATED WORK

It has long been known that processors are designed for complex numerical workloads over primitive data types and are not well-tailored for database workloads. The authors of [23] measured performance not only in terms of response time, but also in terms of hardware performance metrics. They also proposed various modifications to improve the behavior of join algorithms on contemporary processors. Regarding the storage layer, it was soon realized that the established N-ary Storage Model (NSM) penalized execution for the common case of only a small number of fields in each tuple being necessary during query evaluation. This lead to the introduction of vertical partitioning and the Decomposed Storage Model (DSM) [9], where each tuple field is separately stored. This layout reduces the amount of data touched during query evaluation and allows for the use of array computations when implementing the operators. This change of storage layout, however, implied revisiting all query evaluation algorithms. It also affected not only the design of the query engine, but also other orthogonal aspects of a DBMS, *e.g.*, concurrency control. In [2] the Partition Attributes Across, or PAX, storage model was introduced. The idea is that although pages still provide a tuple-level interface, the tuples within a page are vertically partitioned, thus greatly enhancing cache locality. This hybrid approach combines the benefits of NSM and DSM while requiring only moderate changes to the database system.

In the context of the iterator model, a buffering operator was proposed in [25] to increase the tuple granularity in inter-operator communication. This resulted in a measurable reduction in the number of iterator calls across the operator pipeline, but had no effect on the number of evaluation function calls in the body of the operator. In [20] it was proposed that multiple aggregation operations can be combined in a single blocking operator that executes these operations over a sequence of tuples through array computations. Common computations across the aggregation functions are performed only once and stored as intermediate results; array computations are used to evaluate aggregates, a technique more in line with the superscalar design of modern processors.

The state-of-the-art in main-memory query execution is MonetDB [3], [18] where, in addition to vertical decomposition, the entire query engine is built on the notion of array manipulation, with sophisticated query processing techniques (*e.g.*, radix-cluster hash join) having been developed in that context. Though MonetDB's engine employs a different data flow than that of traditional DBMSs, it still is an operator-based approach, tightly connected to the DSM. It also requires materializing all intermediate results, thus reducing the opportunities for exploiting cache locality across separate query operators. These restrictions led to the introduction of MonetDB/X100 [4], [26], where the idea of the blocking operator [20] was coupled with a column-wise storage layout. The use of compound vectorized primitives for performing all computations achieved performance comparable to that of hard-coded implementations.

Prefetching is another area that has received attention, with [7], [8] presenting ways of employing software prefetching in hash join evaluation. Though this approach may improve response times, it introduces the need for dynamically calculating the prefetching distance according to the CPU's frequency, cache latencies, and the run-time load. Inaccuracies result in failing to prefetch the required data on time, or polluting the cache with not immediately needed data. Besides, the cache controller treats software prefetching instructions as hints and may ignore them if there are pending fetch requests. We have therefore chosen not to employ software prefetching in our implementation.

Although a primitive form of code generation was used even in System-R [5], the adoption of the iterator model [11] has dominated query engine design. Code generation was revisited in the Daytona data management system [13], which was capable of on-the-fly generation of query-specific code. It relied, however, on the operating system to perform most

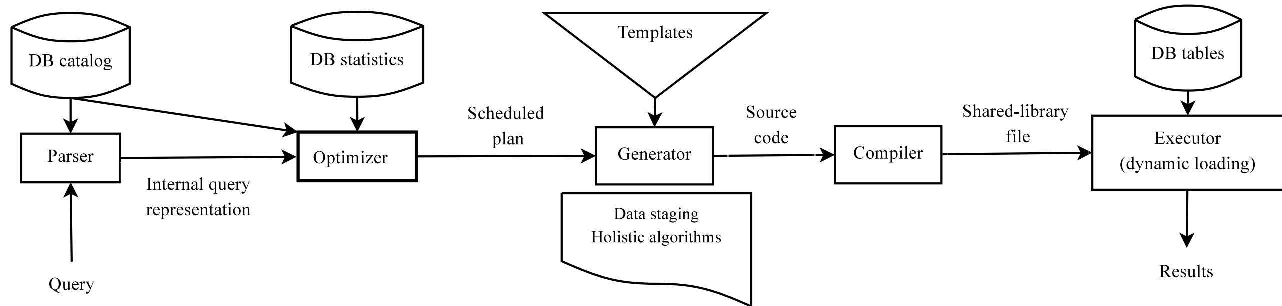


Fig. 2. Holistic query engine overview

of the functionality a DBMS would traditionally provide (*e.g.*, buffering, concurrency control). Similarly, the authors of [21] presented a Java prototype system employing dynamic query compilation. Still, this system employed iterators for operator communication and execution, using the code generator only to remove virtual functions from the body of iterators. Moreover, it did not present any novel query processing options, *e.g.*, joins were exclusively evaluated through preconstructed join indexes.

IV. System overview

In this section we present the design of a query engine employing code generation as the underlying principle for efficient query evaluation. Our system is named HIQUE, standing for *Holistic Integrated Query Engine*. It has been implemented in C/C++ and compiled using the GNU gcc compiler, over the GNU Linux operating system. It adopts the traditional client-server model, *i.e.*, multiple clients communicate with the query engine.

Storage layer. We have adopted the N-ary Storage Model (NSM) as a storage layout, with tuples consecutively stored in pages of 4096 bytes. The system, however, is not tied to the NSM in any way; any other storage model, such as the DSM or the PAX models, can be used and our proposals will still be applicable. Each table resides in its own file on disk, and the system's storage manager is responsible for maintaining information on table/file associations and schemata. A buffer manager is responsible for buffering disk pages and providing concurrency control; it uses the LRU replacement policy. In addition to standard files, the system uses memory-efficient indexes, in the form of fractal B^+-trees [6], with each physical page divided in four tree nodes of 1024 bytes each.

Query processing. The route of a query through the system is shown in Figure 2. The first module is the SQL parser. Our SQL grammar supports conjunctive queries with equi-joins and arbitrary groupings and sort orders. It does not support (*a*) statistical functions in aggregate values, and (*b*) nested queries. We believe, however, these to be straightforward extensions that do not restrict the generality of the holistic evaluation model.

The SQL parser checks the query for validity against the system catalogue and outputs an internal query representation for the optimizer. The latter chooses the optimal evaluation plan using a greedy approach, with the objective of minimizing the size of intermediate results. It also chooses the optimal evaluation algorithm for each operator and sets the parameters used for the instantiation of the code generator's templates (see Section V for more details).

The output of the optimizer is a topologically sorted list O of operator descriptors o_i. Each o_i has as input either primary table(s), or the output of $o_j, j < i$. The descriptor contains the algorithm to be used in the implementation of each operator and additional information for initializing the code template of this algorithm. Effectively, this list describes a scheduled tree of physical operators since there is only one root operator. It is organised so that the first elements describe the joins of the query, followed by any aggregation and sorting operations (unary operators, at most one descriptor for each). The optimizer keeps track of interesting orders [5] and join teams [12], grouping together join operations and avoiding re-sorting where possible. The code generator will then traverse the topologically sorted list and emit a set of functions containing the source code for each operator. This is done in two steps per operator:

1) *Data staging*: all input tables are scanned, all selection predicates are applied, and any unnecessary fields are dropped from the input to reduce tuple size and increase cache locality on subsequent processing. Any pre-processing needed by the following operator, *e.g.*, sorting or partitioning, is performed by interleaving the pre-processing code with the scanning code. The output is then materialized (though not on disk, if the staged input is small enough to fit in main memory).
2) *Holistic algorithm instantiation*: the source code that implements each operator is generated. This code is an instantiation of the appropriate *holistic* algorithm template, as described in Section V-B.

By looking at the inputs of each operator the code generator composes the operator implementations to generate a final function. This function evaluates the entire query and is to be called by the query engine. The final step of the code generation process is to insert all generated functions into a new C source file.

Once the query-specific source code has been generated, a system call invokes the compiler to compile the source file into a shared library file. This step allows the application of aggressive compiler optimizations that target the code of the specific query. The shared library file is then dynamically

Input: 1. Topologically sorted list of operators O,
2. Code templates for data staging (TS),
join evaluation (TJ) and aggregation (TA)
Output: Query-specific C source file

1. **for each** join operator $j_m \in O$
2. retrieve code template $ts_m \in TS$ to stage j_m's inputs
3. **for each** input i_n of j_m
4. instantiate ts_m for i_n
5. generate C function cs_{mn} for staging i_n
6. retrieve code template $tj_m \in TJ$ for j_m's algorithm
7. instantiate tj_m for j_m
8. generate C function cj_m to evaluate join
9. **if** $\exists$ aggregate operator $a \in O$
10. retrieve code template $ts_a \in TS$ to stage a's input
11. instantiate ts_a for a
12. generate C function cs_a for staging a
13. retrieve code template $ta \in TA$ for a's algorithm
14. instantiate ta for a
15. generate C function ca to compute aggregate values
16. **if** $\exists$ ordering operator $s \in O$
17. retrieve code template $ts \in TS$ for sorting
18. instantiate ts and generate sorting C function cs
19. traverse O to compose the function cm calling all functions
20. write all generated functions to a new source file F
21. **return** F

Fig. 3. The code generation algorithm

linked and loaded by the query executor. The latter calls the dynamically loaded function to evaluate the query and redirects the output to the client.

V. CODE GENERATION

In this section we present the implementation of the code generator. The code generator uses a template-based approach. Each algorithm is represented as an abstract template, which is instantiated according to the execution plan.

A. Implementation

The code generator accepts as input the output of the optimizer (*i.e.*, a topologically sorted list O of operator descriptors) and produces a C source file of query-specific code. The generation algorithm is shown in Figure 3. As mentioned, each descriptor contains the algorithm to be implemented, along with the necessary parameters for instantiating code templates. These parameters include the predicate data type(s), information about the operator's inputs, be they primary tables or intermediate results, and the output schema.

Code generation progresses as follows: the generator traverses the operator descriptor list processing the join operators first (Lines 1 to 8 in Figure 3) and moving on to any aggregation (Lines 9 to 15) and ordering operators (Lines 16 to 18). For each operator the generator emits functions that (*a*) stage the input (one function per input table), and (*b*) execute the operator's algorithm. These functions are built by retrieving the appropriate code template (*e.g.*, Lines 2 and 6 for joins) and instantiating it according to the parameters of the operator's descriptor (*e.g.*, Lines 4 and 7). Given that operator descriptors in O contain information about how operators are connected, the last bit of code generation is to traverse O and generate a main (composing) function that calls all evaluation functions in the correct order, ensures the correct (de)allocation of resources and sends the output to the client (Line 19). Finally, all generated functions are put into a new C source file, in the same order as they have been generated.

B. Algorithms and code templates

The goal of holistic algorithms is to use code generation to customize well-known data processing algorithms into more hardware-efficient implementations. Per-query code generation allows the following query-level optimizations: (*a*) attribute types are known *a priori*, which means we can revert separate function calls for data accessing and predicate evaluation to pointer casts and primitive data comparisons, respectively; and (*b*) fixed-length tuples inside each page can be accessed as an array through pointer arithmetic and direct referencing. The system is aware of the differences in latency for accessing each level of the memory hierarchy. Recall from Section II that switching from sequential to random access may even double the latency on accesses outside the D1-cache; moving one layer down the memory hierarchy increases latency by one order of magnitude. The generated code therefore (*a*) examines the input in blocks that fit inside the D1- or the L2-cache, (*b*) maximizes reuse by performing multiple operations over cache-resident data, and (*c*) strives for random access patterns appearing only inside the D1-cache, as this is the only level where the fetch cost is the same for both sequential and random access.

As an example of generated code, Listing 1 shows the C code for filtering the tuples of a table. By employing type information (`int` in this case) and using array accesses, we can eliminate all function calls (but the unavoidable for loading pages and generating the output), saving a large number of CPU cycles. We also reduce the number of instructions executed, as we evaluate predicates over primitive data types. Moreover, the use of array computations allows the code to exploit the processor's superscalar design. The lack of function calls in the inner loop, in combination with directly accessing tuples and their fields by reference, further aids the compiler in optimizing the generated code in ways that efficiently distribute data to registers and favor cache reuse.

Input staging. The staging algorithms include sorting, partitioning, and a hybrid approach. *Sorting* is performed by using an optimized version of quicksort over L2-cache-fitting input partitions and then merging them. Partitioning is either *fine*, through mapping attribute values to partitions, or *coarse*, by using hash and modulo calculations to direct tuples to partitions. *Fine-grained partitioning* is used when the partitioning attribute has a small enough number of distinct values, so that a value-partition map comfortably fits inside the cache hierarchy. For each input tuple, the map is looked up for the corresponding partition. We maintain a sorted array of attribute values and use binary search for lookups. In case the directory spans the lower cache level, searching it may trigger expensive cache misses, so *coarse-grained partitioning* proves more efficient. However, since the generated partitions in the latter case contain multiple attribute values, increasing reuse through

```
1  // loop over pages
2  for (int p = start_page; p <= end_page; p++) {
3   page_str *page = read_page(p, table);
4   // loop over tuples
5   for (int t = 0; t < page->num_tuples; t++) {
6    void *tuple = page->data + t * tuple_size;
7    int *value = tuple + predicate_offset;
8    if (*value != predicate_value) continue;
9    add_to_result(tuple); // inlined
10 }}
```

Listing 1. Optimized table scan-select

```
1  /* Inlined code to hash-partition or sort inputs */
2
3  hash join: // examine corresponding partitions together
4  for (k = 0; k < M; k++) {
5   // update page bounds for both k-th partitions
6   hybrid hash-sort-merge join: // sort partitions
7
8   for (p_1 = start_page_1; p_1 <= end_page_1; p_1++) {
9    page_str *page_1 = read_page(p_1, partition_1[k]);
10   for (p_2 = start_page_2; p_2 <= end_page_2; p_2++) {
11    page_str *page_2 = read_page(p_2, partition_2[k]);
12
13    for (t_1 = 0; t_1 < page_1->num_tuples; t_1++) {
14     void *tuple_1 = page_1->data + t_1 * tuple_size_1;
15     for (t_2 = 0; t_2 < page_2->num_tuples; t_2++) {
16      void *tuple_2 = page_2->data + t_2 * tuple_size_2;
17
18      int *value_1 = tuple_1 + offset_1;
19      int *value_2 = tuple_2 + offset_2;
20      if (*value_1 != *value_2) {
21        merge join: // update bounds for all loops
22        continue;
23      }
24      add_to_result(tuple_1, tuple_2); // inlined
25 }}}}}
```

Listing 2. Nested-loops template for join evaluation

sorting the partitions can help subsequent processing. This leads to a class of what we term *hybrid hash-sort* algorithms, which are applicable in certain join evaluation and aggregation scenarios. These algorithms prove efficient if the number of partitions is big enough so that the largest partition fits in the L2-cache.

Join evaluation. All join evaluation algorithms use the common *nested-loops* template shown in Listing 2; the difference across algorithms lies in how their inputs have been staged. The template uses an array-like sequential access pattern, which favors the utilization of the hardware prefetcher on the first iteration over each page's tuples. Subsequent tuple iterations will be performed over cache-resident pages without any cache misses.

Merge join assumes the input has been previously sorted (Line 1). Join evaluation then progresses by linearly examining the input tables (M is set to 1 in Line 4). The bounds of the loops (both in terms of starting and ending pages per table, and in terms of starting and ending tuples per page) are constantly updated as the merging process progresses (Line 21). This is controlled by a condition variable that can take one of three values: the first means that there is no match between the current tuples; the second means that at least one match has been found and we should continue scanning inner tuples for matches; the last means that the group of inner matching tuples has been exhausted, so we need to advance the outer tuple and backtrack to the beginning of the group of matching inner tuples. If each tuple of the outer loop matches at most once with tuples from the inner loop, the access pattern is linear for both inputs; backtracking to the beginning of a group of matching inner tuples is quite likely to result in cache hits, since small groups will tend to be resident in the L2-, or even the D1-cache.

Partition join builds upon Grace hash join [17]. The input tables are first finely or coarsely partitioned in M partitions each (Line 1); the corresponding partitions (Lines 3 to 5) are then joined using nested-loops join. For fine-grained partitioning, all tuples inside corresponding partitions will match. For coarse-grained partitioning, each tuple will match with none, some, or all tuples of the corresponding partition. To that end, we do not use any hash table as it would lead to uncontrollable random access patterns. Instead, we prefer to first sort the partitions (Line 6) and then use merge join for each pair of corresponding partitions, an algorithm we term *hybrid hash-sort-merge join*. Note that if the size of the partitions is smaller than half that of the L2-cache, by sorting pairs of corresponding partitions just before joining them (instead of during data staging), we ensure that they are L2-cache-resident during join evaluation.

Furthermore, the nested-loops template enables pipelined evaluation of multiple joins without materializing intermediate results, thus radically reducing memory operations and processor stalls. This is applicable in multi-way join queries with join teams, *i.e.*, sets of tables joined on a common key. Such queries are quite common, *e.g.*, in star-schemata or key-foreign key joins. Notions like hash teams and interesting orders, are handled by our model by increasing loop nesting. The template of Listing 2 needs to be slightly modified to support join teams. First, all input tables are properly staged (sorted or partitioned). Then, for each input table the code generator emits one loop over its pages and one over its tuples, with the page loops preceding the tuple loops and following the same table order. The code layout resembles the layout suggested by the loop-blocking code optimization technique [16], which enhances cache locality.

Aggregation. The aggregation algorithms depend on input staging. *Sort aggregation* implies that the input has already been sorted on the grouping attributes. The input is then linearly scanned to identify the different groups and evaluate the aggregate results for each group on-the-fly. For *hybrid hash-sort aggregation*, the input is first hash-partitioned on the first grouping attribute and then each partition is sorted on all grouping attributes. Aggregation then progresses in a single scan of each sorted partition.

Another option is to use value directories for each grouping attribute. This is applicable if the total size of the directories for all grouping attributes is small enough to fit inside the cache hierarchy, so as to avoid directory lookups triggering cache misses. In this case, *map aggregation* is evaluated in a single linear scan of the input without requiring any prior staging. To do so, we maintain one value directory per grouping attribute, as shown in Figure 4(a) for three attributes, and one array per aggregate function holding ag-

value	id
100	0
200	1
300	2

$R.a$

value	id
A	0
B	1
C	2

$R.b$

value	id
Europe	0
Asia	1
Africa	2
America	3

$R.c$

(a) Multiple mapping directories

$$\begin{aligned}&\text{Offset}(R.a = 200, R.b = \text{C}, R.c = \text{Asia})\\ =\ & R.a[200] \cdot |R.b| \cdot |R.c| + R.b[\text{C}] \cdot |R.c| + R.c[\text{Asia}]\\ =\ & 1 \cdot 3 \cdot 4 + 2 \cdot 4 + 1 = 21\end{aligned}$$

(b) Offset of aggregate value

Fig. 4. Group directories for aggregation

gregate values. Assuming that M_i is the map for attribute i and $M_i[v]$ gives the identifier for value v of attribute i, one can then reduce the multi-dimensional mapping of tuple $(v_1, v_2, \ldots, v_n)$, for grouping across n attributes, to the scalar $\sum_{i=1}^{n} \left(M_i[v_i] \prod_{j=i+1}^{n} |M_j|\right)$, where $|M_i|$ is the size of the mapping table for attribute i. The previous formula maps each combination of values to a unique offset in the aggregate arrays, the latter holding $\left(\prod_{i=1}^{n} |M_i|\right)$ values each. An example of applying the formula is shown in Figure 4(b). Aggregate evaluation then progresses as follows: for each input tuple, the grouping attribute maps are used to identify the offset of the group the tuple belongs to. Then, the corresponding variables for this group in each aggregate array are updated with the current values of the aggregate functions.

In all cases, the code generator inlines the code that identifies the groups and applies the aggregate functions. The lack of function calls is particularly important in aggregation: it allows the compiler to generate executable code that widely reuses registers in a computationally-intensive operation. The optimized code minimizes the frequency of interruptions to the execution sequence due to stack interaction, avoids multiple evaluation of common arithmetic expressions, and reduces the number of data accesses per tuple.

C. Development

The main challenges in engineering a code generator for query evaluation were (*a*) the identification of common code templates across different algorithms, (*b*) the interconnection of different operators, since no common interface is present any more, and (*c*) the verification of correctness of the generated code for all supported operations.

The holistic evaluation model eases those problems. The main advantage is that its algorithms exploit generic code templates for all operations. Data staging employs the template of Listing 1; sorting and partitioning operations can be interleaved inside the code. For join evaluation, the nested-loops template of Listing 2 is used in each case, with differences between algorithms either being taken care of through staging, or through extra steps inside the loops. For instance, for hash join, the segments corresponding to Lines 3 to 5 are included and the ones for Lines 6 and 21 are excluded; including the last two code segments will turn the algorithm into hybrid hash-sort-merge join. Aggregation extends the template of Listing 1 by injecting code for tracking different groups and computing the aggregate functions. Furthermore, operators are connected by materializing intermediate results as temporary tables inside the buffer pool and streaming them to subsequent operators.

TABLE I

INTEL CORE 2 DUO 6300 SPECIFICATIONS

Number of cores	2
Frequency	1.86GHz
Cache line size	64B
I1-cache	32KB (per core)
D1-cache	32KB (per core)
L2-cache	2MB (shared)
L1-cache miss latency (sequential)	9 cycles
L1-cache miss latency (random)	14 cycles
L2-cache miss latency (sequential)	28 cycles
L2-cache miss latency (random)	77 cycles
RAM type	2x1GB DDR2 667MHz

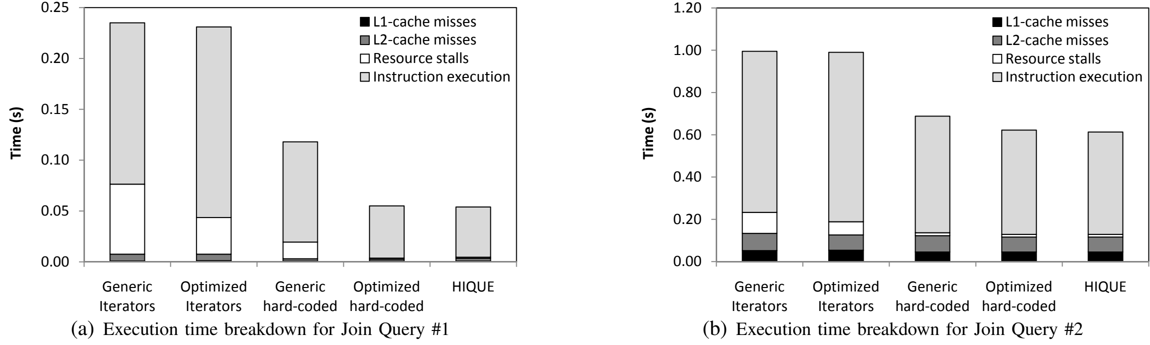
The experience of developing HIQUE has verified these claims. The introduction of new algorithms or even new operators required more effort to extend the parser and the optimizer than to extend the generator. As a general methodology of introducing algorithms, we would first create a model implementation of the new algorithm and compare it to the existing templates. In most cases, the new algorithm resulted in a few different lines of code when compared to the existing evaluation algorithms. We would then extend the templates and the code generator to support the new algorithm. This process was further aided by the output of the code generator being a C source code file: the compiler helped the developer to easily identify errors in the generated code and reduce the number of iterations needed to fully support the new algorithm.

VI. EXPERIMENTAL STUDY

To test the viability of code generation as a general solution to query evaluation we experimented with different aspects of the system. Our aim was to measure (*a*) the superiority of the holistic model over the traditional iterator-based approach, (*b*) the effect of compiler optimizations on the code generated by HIQUE, (*c*) the competitiveness of the system in comparison to other approaches, both research and commercial ones, on established benchmark queries, and (*d*) the penalty for generating, compiling, and linking query-specific code at runtime.

We report results on the currently dominant x86-64 processor architecture. Our system had an Intel Core 2 Duo 6300 dual core processor, clocked at 1.86GHz. The system had a physical memory of 2GB and was running Ubuntu 8.10 (64 bit version, kernel 2.6.27); HIQUE's generated code was compiled using the GNU compiler (version 4.3.2) and with the `-O2` compilation flag. More detailed information about the testing platform can be found in Table I. The cache latencies were measured using the RightMark Memory Analyser [22].

Metrics and methodology. All queries were run in isolation and were repeated ten times each. Each query ran in its own thread, using a single processor core. We did not materialize the output in any case, as the penalty of materialization is similar for all systems and configurations. We report average response times for each system, with the deviation being less than 3% in all cases. We also used hardware performance events as metrics. We obtained the latter with the OProfile [19] tool, which collects sampling data from the CPU's performance event counters. We broke down the execution time into instruction execution, D1-cache miss stalls, L2-cache



(a) Execution time breakdown for Join Query #1

(b) Execution time breakdown for Join Query #2

	CPI	Retired instructions (%)	Function calls (%)	D1-cache accesses (%)	D1-cache prefetch efficiency (%)	L2-cache prefetch efficiency (%)
Generic iterators	0.613	100.00	100.00	100.00	8.33	43.28
Optimized iterators	0.628	91.81	66.99	94.20	10.64	68.35
Generic hard-coded	0.569	53.47	33.87	51.85	27.78	86.84
Optimized hard-coded	0.498	27.63	1.29	39.31	25.00	89.47
HIQUE	0.475	26.22	1.08	36.67	25.00	92.11

(c) Hardware performance metrics for Join Query #1

	CPI	Retired instructions (%)	Function calls (%)	D1-cache accesses (%)	D1-cache prefetch efficiency (%)	L2-cache prefetch efficiency (%)
Generic iterators	0.697	100.00	100.00	100.00	30.67	87.27
Optimized iterators	0.729	95.65	86.86	97.49	30.31	92.20
Generic hard-coded	0.720	67.32	49.56	61.95	60.55	86.38
Optimized hard-coded	0.750	56.80	32.75	56.13	60.95	89.93
HIQUE	0.769	56.62	32.37	54.03	61.07	89.97

(d) Hardware performance metrics for Join Query #2

Fig. 5. Join profiling

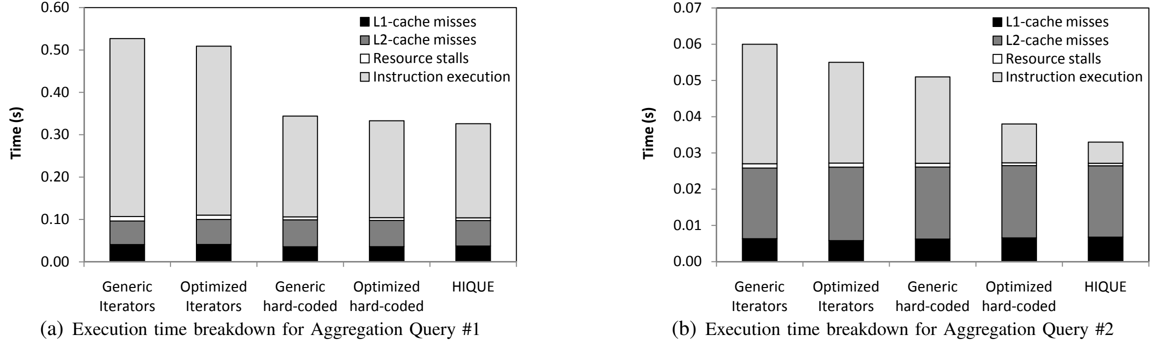
miss stalls and other pipeline resource stalls.[2] To account for hardware prefetching, we assumed sequential access latencies for prefetched cache lines and random access latencies for all other cache misses. This allows for approximate calculation of the cost of cache misses, as the non-blocking design of cache memory allows the CPU to continue executing instructions while fetching data. Still, this methodology provides a good approximation of actual cache miss stall times. In addition to the execution time breakdown, we also calculate the *Cycles Per Instruction* (CPI) ratio, the minimum value being 0.25 for Intel Core 2 Duo processors (*i.e.*, four instructions executed in parallel per CPU cycle). We also measured samples for retired instructions, function calls and D1-cache accesses, normalized to the highest value among the compared configurations for each query. Finally, we report the prefetching efficiency ratio, defined as the number of prefetched cache lines over the total number of missed cache lines.

A. Iterators versus holistic code

To quantify the iterator model's deficiency compared to the proposed holistic model, we compared the following implementations: (*a*) an iterator-based one using generic functions for predicate evaluation, (*b*) a type-specific version of iterators with inlined predicate evaluation, (*c*) a hard-coded implementation using generic functions for predicate evaluation and tuple accesses, (*d*) an improved hard-coded version with direct tuple accesses using pointer arithmetic, and (*e*) the code generated by HIQUE, that further inlines predicate evaluation. We favored the generic implementations by separately compiling the code for each query (including all parameters for instantiating the statically pipelined iterators), thus allowing their extended optimization by the compiler. For join evaluation, we experimented with (*a*) two tables of 10,000 tuples of 72 bytes each using merge-join, with each outer tuple matching with 1,000 inner tuples, and (*b*) two tables of 1,000,000 tuples of 72 bytes each using hybrid-join, with each outer tuple matching with 10 inner tuples. For aggregation, we used a table of 1,000,000 tuples of 72 bytes each, two *sum* functions, and we selected as the grouping attribute one field with either (*a*) 100,000 distinct values, or (*b*) 10 distinct values. We employed hybrid aggregation in the first case and map aggregation in the second. All join and grouping attributes were integers. We used both response times and hardware performance events as metrics. We present the results for join evaluation in Figure 5 and for aggregation in Figure 6.

The first join query is inflationary, as it produces 10,000,000 tuples when joining two tables of 10,000 tuples each. In this case, the nested-loops-based template for join evaluation proves very efficient, as HIQUE is almost five times faster than the iterator implementations. The time breakdown in Figure 5(a) shows that all versions exhibit minimal memory stalls, so the difference in execution time is exclusively due to the lack of function calls, the reduction in retired instructions, and the elimination of resource stalls. Note that the generated code requires 26.22% of the instructions, 36.67% of the data accesses and 1.08% of the function calls when compared to the generic iterator version, as shown in Figure 5(c). Besides, the CPI ratio improves by 22.5% and closes in to the ideal value of 0.25. One can also observe that the efficiency of hardware prefetching more than doubles as the code becomes

[2]Other pipeline resource stalls are defined as resource stalls that are not due to D1- or L2-cache misses – see also [15].



(a) Execution time breakdown for Aggregation Query #1

(b) Execution time breakdown for Aggregation Query #2

	CPI	Retired instructions (%)	Function calls (%)	D1-cache accesses (%)	D1-cache prefetch efficiency (%)	L2-cache prefetch efficiency (%)
Generic iterators	0.796	100.00	100.00	100.00	19.16	94.76
Optimized iterators	0.798	95.35	92.48	99.88	21.73	91.95
Generic hard-coded	0.872	59.85	86.83	91.19	56.79	85.59
Optimized hard-coded	0.875	54.99	77.74	89.32	56.82	86.12
HIQUE	0.919	53.86	68.65	81.63	56.90	88.95

(c) Hardware performance metrics for Aggregation Query #1

	CPI	Retired instructions (%)	Function calls (%)	D1-cache accesses (%)	D1-cache prefetch efficiency (%)	L2-cache prefetch efficiency (%)
Generic iterators	0.791	100.00	100.00	100.00	75.71	95.05
Optimized iterators	0.881	81.85	94.06	74.79	93.18	93.17
Generic hard-coded	0.936	67.62	65.35	60.21	78.93	93.44
Optimized hard-coded	0.904	53.13	32.67	52.72	78.37	95.57
HIQUE	0.899	41.89	4.95	46.13	70.39	95.86

(d) Hardware performance metrics for Aggregation Query #2

Fig. 6. Aggregation profiling

more query-specific, both for the D1- and the L2-cache.

The second join query uses two larger tables as inputs and has much lower join selectivity. In this case, the majority of the execution time is spent on staging the input, *i.e.*, hash-partitioning it and sorting the partitions. Since all versions implement the same algorithm, use the same type-specific implementation of quicksort, and display similar access patterns, the differences in execution times are narrowed. As shown in Figure 5(b) HIQUE is almost twice faster than the iterator-based versions. The penalty for memory stalls is similar in all cases, as expected. The reduction in retired instructions, data accesses and function calls is still significant, according to Figure 5(d), but does not reach the levels of the previous query. Note that the CPI ratio *increases* for hard-coded versions. This is due to the retirement of fewer instructions in total, so the contribution of costly memory operations to the CPI is more substantial. Prefetching efficiency doubles for the D1-cache and is approximately 90% for the L2-cache in all cases.

In terms of aggregation, the first benchmark query was evaluated using the hybrid hash-sort algorithm. In this case staging dominates execution time, as aggregation is evaluated in a single scan of the sorted partitions. Still, as shown in Figure 6(a), HIQUE maintains an advantage of a factor of 1.61 over iterators. The use of the same partitioning and sorting implementations leads to similar memory stall costs for all code versions. The difference in execution times mainly stems from the reduction in instructions, data accesses and function calls, according to Figure 6(c). Observe that the efficiency of the D1-cache prefetcher increases three times, while that of the L2-cache reaches almost 90% for all implementations.

In the case of the proposed map-based algorithm, aggregation is evaluated in a single pass of the input without any need for intermediate staging. This allows the code generator to inline all group tracking and aggregate calculations in a single code segment. As shown in Figure 6(b), the code generated by HIQUE outperforms generic iterators by almost a factor of two. Memory stalls dominate execution time for the HIQUE version (though their effect might be alleviated from the operation of non-blocking caches), as the aggregate calculations require only a few instructions per tuple. Also shown in Figure 6(b), the reduction in function calls is gradual as the code becomes more query-specific and reaches 4.95% for the most optimized hard-coded version. Furthermore, the linear scan of the input helps the hardware prefetchers achieve high levels of efficiency, over 70% for the D1-cache and near 95% for the L2-cache in all cases.

We next examined the efficiency of compiler optimizations on the iterator-based and the hard-coded implementations. We compiled the various implementations with compiler optimizations disabled (by setting the optimization flag to `-O0` for the GNU compiler) and ran the same join and aggregation queries. The results are presented in Table II. Naturally, the differences between the various code versions are more tangible when there are no compiler optimizations, since the compiler can apply some of the optimizations that are included in the code generation process. For example, the compiler may inline the functions for predicate evaluation, so the differences between the last two implementations are narrowed in all queries, but become apparent when the `-O0` optimization flag is used.

The results show that compiler optimizations are most efficient in the first join query, resulting in speedups between 2.67 and 4.85, as the loop-oriented code transformations can

TABLE II

EFFECT OF COMPILER OPTIMIZATION (RESPONSE TIMES IN SECONDS)

	Join Query #1		Join Query #2		Aggregation Query #1		Aggregation Query #2	
	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2
Generic iterators	0.802	0.235	1.953	0.995	1.225	0.527	0.136	0.060
Optimized iterators	0.618	0.231	1.850	0.990	1.199	0.509	0.113	0.055
Generic hard-coded	0.430	0.118	1.421	0.688	0.586	0.344	0.095	0.051
Optimized hard-coded	0.267	0.055	1.225	0.622	0.554	0.333	0.080	0.038
HIQUE	0.178	0.054	1.138	0.613	0.543	0.326	0.070	0.033

improve performance on iterative tuple processing. For the rest of the queries the speedup is almost a factor of two. Since we compile the code for each query and for all implementations, the speedup is significant even for the iterator-based ones. Moreover, the compiler is less efficient on the hard-coded implementations: the source code is already minimalistic and contains various optimizations (*e.g.*, loop blocking, function inlining). Still, the simplicity of the code and the lack of function calls allows the compiler to further improve the hard-coded versions resulting in significant speedups.

B. Performance of holistic algorithms

We now move on to examine the performance of the proposed algorithms while varying the characteristics of the input and the predicates to be applied. We compared the optimized iterator-based versions of the proposed algorithms with the code HIQUE generates for each query. In Figure 7(a) we examine scalability in join evaluation. We used two tables with a tuple size of 72 bytes. Each outer tuple matched with ten inner tuples on integer join attributes. The cardinality of the outer table was set to 1,000,000, while the inner one's varied between 1,000,000 and 10,000,000. The results show that all algorithms scale linearly, with iterator-based hash-sort-merge-join having similar performance to HIQUE's merge-join. As expected, the generated version of the hash-sort-merge join outperforms all other versions by a substantial margin, proving its efficiency in a wide range of input cardinalities.

In multi-way queries, the evaluation of multiple joins using a single segment of deeply-nested loops improves performance as the generated code does not require materialization of intermediate results. To verify this, we joined one table of 1,000,000 tuples with a varying number of tables of 100,000 tuples each, on a single join attribute. All tables had 72-bytes-sized tuples, while the output cardinality was 1,000,000 in all cases. We compared the binary iterator-based merge-join, its equivalent when generated by HIQUE, and the code versions when join teams where enabled in HIQUE, using either merge- or hybrid-join. The results of Figure 7(b) show that although iterator-based merge-join takes advantage of sorted orders, it is widely outperformed by its holistic equivalent. Furthermore, the adoption of join teams radically reduces execution time, with the difference between HIQUE and iterators reaching a factor of 3.32 when joining eight tables. The extension of the nested-loops join template to support join teams therefore pays off in the case of multi-way join queries.

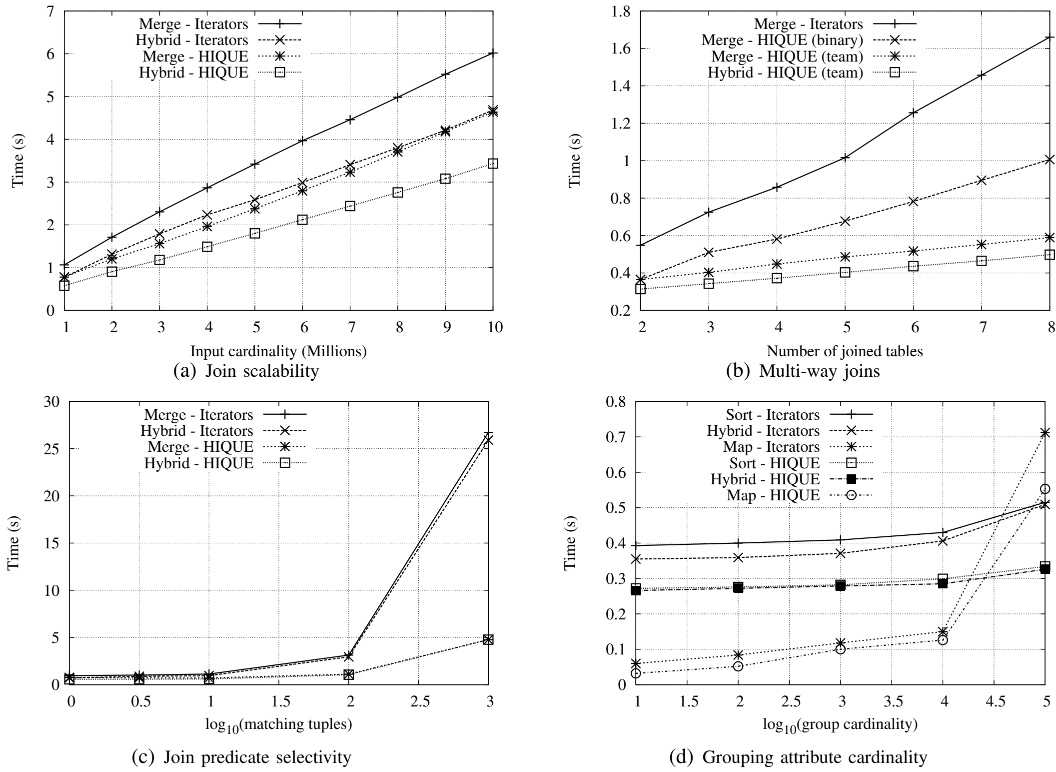
Highly-selective join predicates are expected to increase the difference in performance between the iterator and the holistic model. This is due to the number of iterator calls growing larger and the join evaluation cost surpassing that of input staging; the latter cost is similar for all implementations. This is shown in Figure 7(c) for joining two tables of 1,000,000 tuples each. Each input tuple was 72 bytes wide, while the number of inner matching tuples per outer tuple varied between 1 and 1,000. The results show that the gap between the iterator-based and the holistic implementations widens quickly as join selectivity increases and reaches a factor of five for 1,000 matches per outer tuple.

The salient factor in aggregation performance is the domain of the grouping attribute(s). If this domain allows the value directories and the aggregate arrays (see also Section V-B) to simultaneously fit in the lowest cache level, map aggregation is expected to outperform the algorithms that require input staging. We show the effect of the grouping attribute's range in Figure 7(d). The input table had 1,000,000 tuples of 72 bytes each. We used two *sum* functions and one grouping attribute as we varied the number of distinct values between 10 and 100,000. The results verify that, for small numbers of groups, map aggregation is highly efficient, both in its iterator-based and its holistic form. However, sort and hybrid aggregation are only moderately affected by the number of groups. They perform better than map aggregation when the auxiliary data structures of the latter (*i.e.*, the value directory for the grouping attribute and the aggregate arrays) span the L2-cache, the difference approaching a factor of two for 100,000 groups.

C. TPC-H benchmark

The last set of experiments is over the standard and more realistic TPC-H benchmark [24]. We benchmarked HIQUE against three database systems: (*a*) PostgreSQL (version 8.2.7), a widely-used and high-performance open-source DBMS over NSM that employs the iterator model, (*b*) a commercial system, which we refer to as *System X* for anonymity, also using NSM and iterators but employing software prefetching instructions to reduce cache miss stalls, and (*c*) MonetDB (version 5.8.2), an architecture-conscious DBMS that uses a DSM-based storage layer and column-wise evaluation algorithms. This choice allowed the comparison of different storage systems and query engine designs, with PostgreSQL representing the traditional I/O-optimized design, System X bridging the gap between I/O- and CPU-bound execution with software prefetching, and MonetDB being a design optimized for main-memory execution.

We used the benchmark's generator to generate a data-set with a scaling factor of one. The tables were not altered in any way (*e.g.*, sorted) before being imported to the systems. The "raw" data-set size was approximately 1.3GB, without indexes, thus fitting in our system's main memory. We built indexes in all systems, gathered statistics at the highest level of detail, and set the memory parameters to allow in-memory



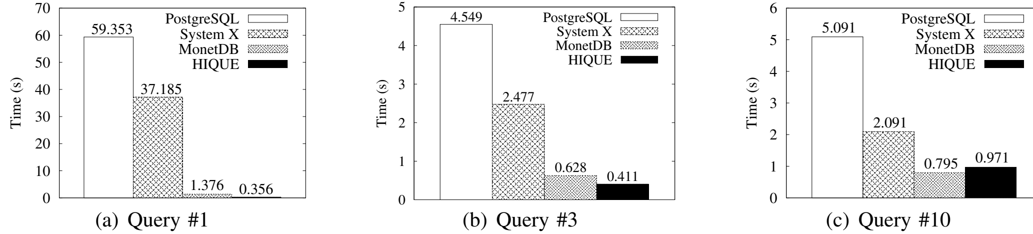
(a) Join scalability

(b) Multi-way joins

(c) Join predicate selectivity

(d) Grouping attribute cardinality

Fig. 7. Join and aggregation performance



(a) Query #1

(b) Query #3

(c) Query #10

Fig. 8. TPC-H Queries

execution. We experimented with TPC-H Queries 1, 3 and 10. These include highly selective join predicates that cannot be evaluated as join teams, as well as aggregation operations of a varying number of grouping attributes and aggregate functions. TPC-H tables have wide tuples spanning multiple cache lines, with only a few fields actually needed by any query. Thus, the expectation is for MonetDB to benefit from vertical partitioning and outperform all NSM-based systems.

TPC-H Query 1 is an aggregation over almost the entire `lineitem` table (about 5,900,000 tuples) and produces four output groups. As the two aggregation attributes have a product of distinct value cardinalities equal to six, the most appropriate holistic aggregation algorithm is map aggregation. The results in Figure 8(a) show HIQUE outperforming MonetDB by a factor of four and the other NSM-based systems by two orders of magnitude, reaching a 167-fold advantage over PostgreSQL. This is due to the holistically generated code: it inlines all selection, grouping, and aggregation operations in a single succinct code block that lacks function calls and is tailored towards efficient register utilization. The measured performance translates to 662.16 millions of CPU cycles, which is comparable to that of MonetDB/X100's DSM-based approach and 30% faster than MonetDB/X100's NSM-based approach [26]. Hence, we posit that HIQUE generates code that is identical to a hard-coded implementation, thus achieving maximum efficiency in aggregation (at least for NSM-based systems).

The remaining queries test join evaluation, aggregation, and sorting. The holistic optimizer stages all inputs before further operations. This is expensive over the benchmark tables, as only a small portion of each tuple is used by the query operators. The queries are a perfect match for DSM systems, like MonetDB: through vertical partitioning only the required fields for each operator are fetched. As a result HIQUE is 34.5% faster and 18.1% slower in Queries 3 and 10 respectively, when compared to MonetDB. Compared to the NSM systems, HIQUE outperforms PostgreSQL and System X by a substantial factor, ranging between 2.2 and 11.1.

The TPC-H results prove the viability of holistic evaluation in a realistic query workload. The holistic model provides code simplicity and enhances cache locality during execution, there-

TABLE III

QUERY PREPARATION COST

TPC-H Query	SQL processing (ms)			Compilation (ms)		File sizes (bytes)	
	Parse	Optimize	Generate	with -O0	with -O2	Source	Shared library
#1	21	1	1	121	274	17,733	16,858
#3	11	1	2	160	403	33,795	24,941
#10	15	1	4	213	619	50,718	33,510

fore reducing the number of instructions and data accesses required to evaluate queries. That way, both the processor and the memory subsystem are stressed to a lower extent, leading to a significant speedup of query evaluation. This allowed our NSM-based system to achieve performance that was so far conceivable only for systems employing vertical partitioning.

D. Code generation cost

The drawback of per-query code generation is the overhead for emitting and compiling query-specific source code. To quantify this cost we report in Table III the preparation times for the TPC-H queries. We separately show the query parsing, optimization, code generation and compilation times, as well as the sizes of the generated source and shared-library files. The time for parsing, optimizing, and generating code is trivial (less than 25ms). Compiling the generated code takes longer and compilation time depends on the optimization level. Compilation takes 121–213ms with no optimizations (`-O0` compiler flag), but needs 274–619ms when the optimization level is increased (`-O2` compiler flag). The generated source and shared library file sizes are less than 50 kilobytes.

Preparation time is not negligible and can be a significant portion of execution time for short-running queries. In such cases it is preferable to avoid applying compiler optimizations that increase compilation time; the gain in execution time will be intangible. Additionally, it is common for systems to store pre-compiled and pre-optimized versions of frequently or recently issued queries. This is certainly applicable in HIQUE, especially if we take into account the small size of the generated binary files. Besides, in most cases the performance benefits outweigh the generation cost.

VII. CONCLUSIONS AND FUTURE WORK

We have presented the case for holistic query evaluation. The proposed technique is based on generating query-specific code that integrates multiple query operations in succinct code constructs. The generation process uses code templates for each query operator and builds query-specific code with the following objectives: (*a*) minimum function calls, (*b*) reduced instructions and memory accesses, and (*c*) enhanced cache locality. The proposed model exhibits a substantial performance advantage when implemented over the NSM-based storage layer. It also does not affect any orthogonal aspects of a DBMS like concurrency control and recovery. To verify the advantages of the proposed holistic model, we implemented HIQUE — the Holistic Integrated Query Engine. Extensive experiments with a variety of data-sets and query workloads proved HIQUE's potential for per-query code generation, verifying the efficiency of our approach in main-memory execution.

The next step is to extend our approach for multithreaded processing. Current processor designs integrate multiple cores sharing the lowest on-chip cache level. Though this design widens opportunities for parallelism, it introduces resource contention. We believe that code generation is advantageous for such designs: one can accurately specify the code segments that can be executed in parallel, thus reducing synchronization overhead and memory bandwidth requirements.

REFERENCES

[1] Anastassia Ailamaki et al. DBMSs on a Modern Processor: Where Does Time Go? In *The VLDB Journal*, 1999.

[2] Anastassia Ailamaki et al. Weaving Relations for Cache Performance. In *The VLDB Journal*, 2001.

[3] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, 2002.

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

[5] Donald D. Chamberlin et al. A history and evaluation of System R. *Commun. ACM*, 24(10), 1981.

[6] Shimin Chen et al. Fractal prefetching B+-Trees: optimizing both cache and disk performance. In *SIGMOD*, 2002.

[7] Shimin Chen et al. Improving hash join performance through prefetching. In *ICDE*, 2004.

[8] Shimin Chen et al. Inspector Joins. In *VLDB*, 2005.

[9] George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[10] Jack Doweck. Inside Intel Core Microarchitecture and Smart Memory Access, 2005. White paper.

[11] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2), 1993.

[12] Goetz Graefe et al. Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB*, 1998.

[13] Rick Greer. Daytona And The Fourth-Generation Language Cymbal. In *SIGMOD*, 1999.

[14] John Hennessy and David Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2006.

[15] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2008.

[16] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[17] Masaru Kitsuregawa et al. Application of Hash to Data Base Machine and Its Architecture. *New Generation Comput.*, 1(1), 1983.

[18] S. Manegold et al. What happens during a Join? - Dissecting CPU and Memory Optimization Effects. In *VLDB*, 2000.

[19] OProfile. A System Profiler for Linux, 2008. `http://oprofile.sourceforge.net/`.

[20] Sriram Padmanabhan et al. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, 2001.

[21] Jun Rao et al. Compiled Query Execution Engine using JVM. In *ICDE*, 2006.

[22] RightMark. RightMark Memory Analyser, 2008. `http://cpu.rightmark.org/products/rmma.shtml`.

[23] Ambuj Shatdal et al. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*, 1994.

[24] Transaction Processing Performance Council. The TPC-H benchmark, 2009. `http://www.tpc.org/tpch/`.

[25] Jingren Zhou and Kenneth A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD*, 2004.

[26] Marcin Zukowski et al. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN*, 2008.