# Efficiently Compiling Dynamic Code for Adaptive Query Processing

Tobias Schmidt
TU Munich
tobias.schmidt@in.tum.de

Philipp Fent
TU Munich
fent@in.tum.de

Thomas Neumann
TU Munich
neumann@in.tum.de

## ABSTRACT

Query optimization is one of the key challenges in modern database systems. With sophisticated data processing techniques like query compilation or vectorization, a good execution plan is essential for high performance. Yet, finding the optimal implementation for a query upfront is difficult as optimizers must rely on cardinality estimations and coarse cost models for the underlying hardware. Adaptive Query Processing overcomes these limitations: We evaluate different implementations for the same query during execution and choose the best-performing implementation based on accurate runtime measurements.

However, compiling database systems cannot easily modify the generated code, and recompiling queries is prohibitively expensive. We propose a novel compilation technique — Dynamic Blocks — that avoids recompilations by embedding code fragments for all variants into the generated code. We integrate the approach into our research system Umbra and implement two dynamic optimizations for reordering joins and adapting selections during execution. Our results show that Adaptive Query Processing improves the runtime of data-centric code by more than 2×.

## 1 INTRODUCTION

Hardware trends like the increase of main-memory sizes and the ever-growing number of execution threads per machine paved the way for new data processing techniques and allows for more complex data processing tasks and workloads. In addition, different hardware platforms can exhibit diverse performance characteristics that affect the execution times. Nevertheless, database systems rely on heuristic and coarse cost models to optimize and evaluate queries. For example, the cache sizes, memory access speed, and instruction throughput vary dramatically from small and lightweight devices to high-performance server processors. Hence, query engines are susceptible to data skew and misestimations, which can, for example, result in suboptimal join orders [26]. Taking all these factors into consideration and finding the optimal implementation for a query in advance is challenging.

High-performance data processing workloads, therefore, require more adaptive techniques that postpone optimization decisions and make them at run-time when accurate statistics and cost information are available [35]. Adaptive Query Processing (AQP) techniques can choose different implementations for evaluating relational operators or expressions while executing a query and change the execution order of filter predicates and joins [4, 12, 38]. Besides the potential performance improvements, adaptive processing allows the execution engine to react to the underlying hardware platform and changing characteristics in the workload.

**Example:** Consider the following query that filters the `lineitem` relation, similar to Q12 of the TPC-H benchmark, using two predicates and counts the number of qualifying tuples:

```
select count(*)
from lineitem
where l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
```

Although the query looks simple and consists of few operations (a table scan, two predicates, and one aggregation), finding the optimal implementation is not that easy. Figure 1 shows three possible variants for evaluating the filter condition. The first two implementations branch after each predicate but with different evaluation orders, and the third variant evaluates both conditions together.

Variant Ⓐ is the result of the commonly used heuristic to order predicates by selectivity. Since we already filter many tuples after the first predicate, we load the least amount of data from the base table and execute the fewest instructions. However, this comes at the cost of frequent CPU branch mispredictions [39]. Table 1 shows these performance characteristics for each of the variants.

While Ⓐ does less work, the total execution time is highest among all implementations due to branch misses. Finding a heuristic that covers all cases is almost impossible. It requires accurate statistics for each predicate, which is already challenging due to correlations [19] and precise estimates of the number of executed instructions and cycles. For instance, string comparisons or `like` predicates execute far more instructions, and branching after each condition might be beneficial again.

Most query execution engines either fall back to the simple heuristic that orders the predicates by selectivity and branches immediately or use vectorized execution with a higher instruction overhead for complex predicates [9]. Our example illustrates that the execution time — our ultimate optimization goal — has non-trivial and hardware-dependent connections to the generated code. While simple heuristics might work for many queries, they already fail for moderate challenges like our example, where Ⓐ is 1.5× slower than Ⓒ. As shown by Dreseler et al., evaluating predicates is one of the major choke points in the TPC-H [13]. To confidently find the best implementation for any query, we need to gather empiric measurements instead of blindly trusting a heuristic. With

 To Appear in the 13th Workshop on Accelerating Analytics and Data Management (ADMS'22), September 2022, Sydney, Australia.

```
for l in lineitem:
  if not l_shipdate < l_commitdate:
    continue  -- 51% taken
  if not l_commitdate < l_receiptdate:
    continue  -- 75% taken

  counter++
```

**Variant Ⓐ: Separate branches**

```
for l in lineitem:
  if not l_commitdate < l_receiptdate:
    continue  -- 37% taken
  if not l_shipdate < l_commitdate:
    continue  -- 81% taken

  counter++
```

**Variant Ⓑ: Separate branches**

```
for l in lineitem:
  if not (l_shipdate < l_commitdate
     and l_commitdate < l_receiptdate):
    continue  -- 88% taken

  counter++
```

**Variant Ⓒ: Combined branch**

**Figure 1: Three different implementations for filtering tuples in a table scan on `lineitem`. Standard query optimizers will use Ⓐ as it processes the least tuples. However, depending on the processed data and the underlying hardware platform, Ⓑ or Ⓒ perform better.**

**Table 1: Performance characteristics end the overall execution time of the three implementations from Figure 1 on TPC-H SF 10 (the hardware counters are normalized to the number of scanned tuples).**

| Variant | branch-misses | instructions | loads | exec. time |
|---|---|---|---|---|
| Ⓐ | 0.63 / tpl | 7.62 / tpl | 2.85 / tpl | 18.4 ms |
| Ⓑ | 0.58 / tpl | 7.91 / tpl | 3.00 / tpl | 17.7 ms |
| Ⓒ | 0.13 / tpl | 11.67 / tpl | 3.37 / tpl | **12.7 ms** |

AQP, we collect and evaluate these statistics during execution and avoid system-dependent bias.

Micro Adaptivity [35] and Permutable Compiled Queries [29] recently demonstrated the practicality of AQP in modern database systems. Both techniques follow the same principle: vectorized query engines use so-called primitives to collectively process a block or vector of tuples. Changing the function pointers to the primitives during execution makes it possible to choose different implementations and adapt the query plan at run-time. During execution, both systems evaluate different variations of the plan and choose the best-performing implementation to process most of the data. However, this limits the execution model to vector-at-a-time processing since calling primitives for every tuple results in a significant performance overhead. Compiling database systems, like HyPer [21], Amazon Redshift [2], Hyrise [14], or Umbra [31], must recompile the query plan to change the operators' implementations.

In our paper, we propose a novel query compilation technique called *Dynamic Blocks* that avoids recompilations and generalizes Adaptive Query Processing to any compiling database system, especially also with tuple-at-a-time processing. Our approach allows exchanging operator implementations, reordering code fragments, and efficiently generates optimized machine code during query execution. Different dynamic block types augment a program's control flow before execution but after compiling it. Developers can use this mechanism to specify different implementations for the same query and choose one implementation later.

To demonstrate the practicality of our technique, we integrated it into the research database system Umbra [31] and implemented two adaptive optimizations for reordering joins and adapting selections. Umbra's execution engine first explores different variants of a query and records their runtime. Afterward, the best-performing implementation is used, allowing us to find the optimal query execution strategy based on accurate measurements. Our experiments show that the Dynamic Blocks framework improve execution times by up to 2×, with almost no overhead and minor performance penalties.

The rest of this paper is organized as follows: We first give a brief overview of prior work on AQP in database systems before introducing our novel Dynamic Blocks framework and its integration into Umbra's existing code generation process in Section 3. Section 4 discusses how Umbra finds the optimal variant during execution and exploits it. Afterward, Section 5 introduces two adaptive optimizations and discusses their implementation using Dynamic Blocks. In Section 6, we evaluate the new framework and the benefits of AQP on four analytical benchmarks.

## 2 RELATED WORK

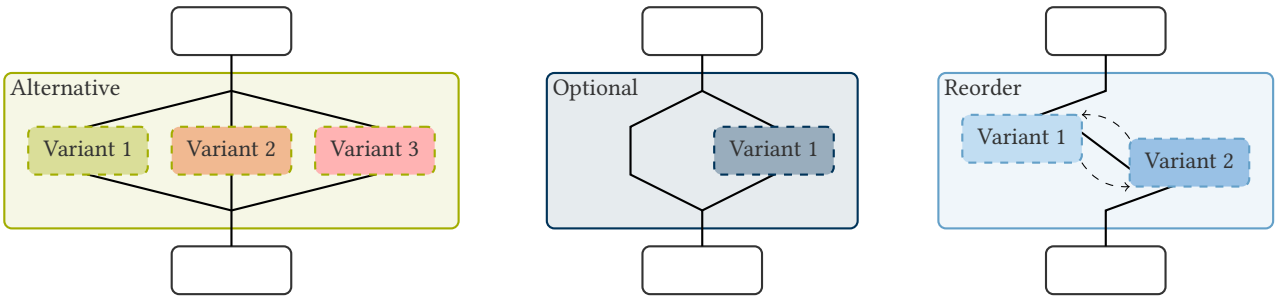
AQP dates back to the Ingres database's one-variable query processor [43]. Since then, new approaches have been proposed, and several surveys were conducted [11, 12, 18]. To put our technique into contrast, we rely on the taxonomy of AQP systems of Babu and Bizarro [4]:

**Plan-based systems** re-optimize the query execution plan if the observed behavior deviates from the estimations. We further distinguish two implementation styles in these systems: (a) *Mid-query Re-optimization* uses the query optimizer to generate a new (better) plan during query execution, and (b) *Parametric Query Optimization* initially builds multiple optimal plans for different situations and chooses the best plan at run-time.

Re-optimizing systems adapt the query plan while executing it. The query optimizer identifies key points in the plan with high inaccuracy potential and collects statistics on data distribution and intermediate result sizes. The system changes the plan if the gathered statistics differ significantly from the initial estimates [20]. The effectiveness of *Mid-query Re-optimization* has been demonstrated in both SQL Server [32] and PostgreSQL [34].

*Parametric Query Optimization* prepares a set of optimal candidate plans for different parameters like predicate selectivities, memory usage, or user input. A special *choose-plan* operator selects the best candidate at run-time and executes it [6, 8, 15, 17]. A major drawback of this technique is the expensive analysis for finding the set of plans as the search space grows exponentially with the number of parameters.

**Routing-based systems** delegate the decision of how to process tuples to the runtime system. The optimizer inserts special, adaptive operators into the plan that can evaluate each tuple independently




**(a) Alternative block** **(b) Optional block** **(c) Reorder block**

**Figure 2: Dynamic Blocks allow intricate control flow changes at runtime.**

using different implementations or execution orders. This technique avoids re-invocations of the query optimizer, and can adapt operators and execution pipelines while evaluating them [18].

Avnur and Hellerstein introduced the adaptive *eddy* operator to change the execution order of pipelined operators. It adaptively re-routes the tuples during execution to different operators. Several systems have adopted this technique to correct misestimation and react to changing data distributions [10, 36, 41].

**Modern systems** of the last decade shifted the execution paradigm from Volcano-style iteration [16] with tuple-at-a-time processing to vectorized execution [7] or compilation [30]. This made it difficult to hide the performance overhead of routing-based AQP. Nevertheless, Raducanu et al. and Perron et al. demonstrated the practicality of tuple-routing techniques in combination with vectorization: Both systems, VectorWise and NoisePage, adapt the execution not for a single tuple but for an entire block of hundreds of tuples [29, 35]. Rosenfeld et al. used the Micro Adaptivity approach for adaptively optimizing aggregations and selections on heterogeneous devices including GPUs [38].

However, compiling database systems with tuple-at-a-time processing, like HyPer or Umbra, cannot easily employ AQP without compiling query plans multiple times. Our Dynamic Blocks compilation framework tackles exactly this problem: It generates dynamic code that can be adapted at run-time without sacrificing performance or recompiling queries. As the Permutable Compiled Queries technique in NoisePage [1], our approach supports routing-based optimizations that adapt implementations or the execution order of pipelined operators.

## 3 DYNAMIC BLOCKS

Tuple-routing techniques adapt relational operators or pipelines while executing them. Dynamic Blocks generalize routing-based AQP to compiling database systems while avoiding recompilations. As stated in the introduction, we use our research system Umbra [31], which generates code according to the producer-consumer model and uses data-centric processing [30]. Unlike NoisePage or VectorWise, Umbra evaluates the entire pipeline for each tuple, minimizing tuple materialization points and keeping values in machine registers (or on the stack). Hence, existing AQP techniques from vectorized query engines would introduce expensive function calls on a per-tuple basis. To maintain the data-centric approach, we will not manipulate function pointer arrays but adapt the generated code itself. The Dynamic Blocks framework generates code fragments for all variants of a query and reorders or exchanges them during execution. This approach enables both high-level plan changes and low-level optimizations, like reordering operators or switching between filter implementations.

In the following section, we first describe the centerpiece of our novel code generation technique, the so-called *Dynamic Blocks*. Afterward, we discuss their integration into Umbra's existing compilation process and custom intermediate representation. Finally, Section 3.3 presents an optimized compiler for Dynamic Blocks.

### 3.1 Block Types

Dynamic Blocks follow the idea of generating linear code for all options. Instead of compiling new code for every variation of a pipeline, e.g., the three code snippets from Figure 1, we embed all variants into a single program to avoid duplicate work for code generation and unnecessary recompilations. Existing code generation frameworks like LLVM [24] or Umbra's Tidy Tuples [22] lack the abstraction to describe the different implementation alternatives. We, therefore, propose three types of Dynamic Blocks that augment the control flow before executing a program but after compiling it. Figure 2 illustrates the semantics of the available block types:

**Alternative Blocks** choose between different, equivalent implementations. Each alternative sub-block contains the code for one of the possible variants.

**Optional Blocks** adaptively enable or disable the code in the variant during execution.

**Reorder Blocks** can be used to change the evaluation order of commutative logic, e.g., join probes or predicates. They reorder the contained code fragments, resulting in $n!$ variations of the function.

With Dynamic Blocks, we can merge the three implementations from Figure 1 into a single program in Figure 3. While the table scan in line 1 and the group by operator in line 10 are compiled, as usual, we now place the filter predicates in a dynamic block. In the top-level variants, we distinguish between the single-branch implementations (Variant 1) and the versions with a branch after each predicate (Variant 2). The second variant then contains another nested dynamic block that changes the evaluation order of the filters during execution. We use a reorder block where the Variant 2.1

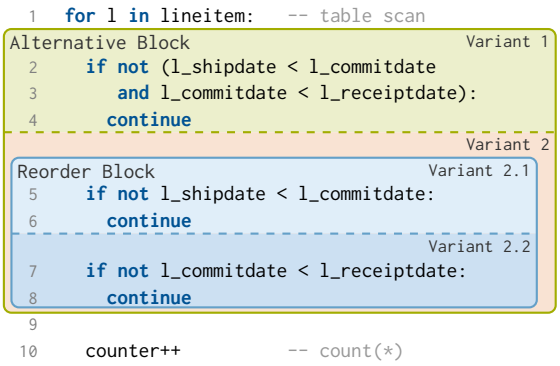



**Figure 3: Adaptive implementation with Dynamic Blocks for the example query and its three implementations.**

evaluates the predicate `l_commitdate < l_receiptdate`, and Variant 2.2 filters the tuples using `l_shipdate < l_commitdate`. This example also showcases the composability of our Dynamic Blocks: We can combine and nest them arbitrarily deep and still integrate them into the regular surrounding code.

While these three block types in isolation offer only limited functionality, a combination of multiple blocks is flexible enough to express many adaptive optimizations. In Section 5, we discuss applications of Dynamic Blocks that integrate well into the existing code generated by Umbra. In addition, all block types can easily be lowered to Umbra's internal code representation.

## 3.2 Integration into Umbra's Intermediate Representation

To transform the high-level concept of Dynamic Blocks into dynamic code, we need to represent them in the code our system generates. Umbra first translates the query execution plan into a custom intermediate representation (IR) and then lowers the generated code to an executable program. Umbra's IR is strongly influenced by LLVM's intermediate representation [24] and can be described as a subset with database-specific operations. Several aspects are identical, including using the static single assignment form [37] and dividing the program into basic blocks based on terminating instructions (e.g., branches or returns). Especially the SSA form simplifies lifetime analysis and facilitates optimizations such as constant propagation or dead code elimination.

We now connect the Dynamic Blocks from Section 3.1 to Umbra's IR. As mentioned before, a dynamic block consists of one or more variants, which contain code fragments for adapting the execution. To integrate well with the existing IR, we always assign entire basic blocks to variants and only mark the transitions between these as dynamic with a `dynbr` instruction. Since it replaces regular block transitions, this representation allows Dynamic Blocks to contain regular basic blocks and other dynamic blocks, e.g., to represent the nesting of Variant 2.1 from Figure 3. We maintain this connection between high-level blocks and low-level IR in two additional management structures: One that tracks the Dynamic Blocks and their variants and a second structure that maps basic and dynamic blocks to variants.

```
1  ...
2  .loopTuples:
3    %l_commitdate_data = getelementptr int8 %data, i64 3932160
4    %l_commitdate = load int32 %l_commitdata_data, %localTid
5    dynbr alternative .enterAlternativeBlock .leaveAlternativeBlock
6
Alternative Block                                              Variant 1
7  .enterAlternativeBlock:
8    %l_shipdate_data = getelementptr int8 %data, i64 3670016
9    %l_shipdate = load int32 %l_shipdate_data, %localTid
10   %predicate1 = cmpult i32 %l_shipdate, %l_commitdate
11   %l_receiptdate_data = getelementptr int8 %data, i64 4194304
12   %l_receiptdate = load int32 %l_receiptdate_data, %localTid
13   %predicate2 = cmpult i32 %l_commitdate, %l_receiptdate
14   %condition = and bool %predicate1, %predicate2
15   condbr %condition .continue .continueScan
16
17 .continue:
18   dynbr alternative .nextVariant .leaveAlternativeBlock
19
20 .nextVariant:                                                 Variant 2
21   dynbr reorder .enterReorderBlock .leaveReorderBlock
22
Reorder Block                                                  Variant 2.1
23 .enterReorderBlock:
24   %l_shipdate_data1 = getelementptr int8 %data, i64 3670016
25   %l_shipdate1 = load int32 %l_shipdate_data1, %localTid
26   %condition1 = cmpult i32 %l_shipdate1, %l_commitdate
27   condbr %condition1 .continue_1 .continueScan
28
29 .continue_1:
30   dynbr reorder .nextVariant_1 .leaveReorderBlock
31
32 .nextVariant_1:                                               Variant 2.2
33   %l_receiptdate_data1 = getelementptr int8 %data, i64 4194304
34   %l_receiptdate1 = load int32 %l_receiptdate_data1, %localTid
35   %condition2 = cmpult i32 %l_commitdate, %l_receiptdate1
36   condbr %condition2 .continue_2 .continueScan
37
38 .continue_2:
39   dynbr reorder .leaveReorderBlock .leaveReorderBlock
40
41 .leaveReorderBlock:
42   dynbr alternative .leaveAlternativeBlock .leaveAlternativeBlock
43
44 .leaveAlternativeBlock:
45   ...
46 .continueScan:
47   ...
```

**Figure 4: Dynamic IR code for evaluating the filter predicates from Figure 3 with Dynamic Blocks generated by Umbra.**

Figure 4 shows the generated code for our example in Umbra's IR. We only show the relevant part responsible for adaptively evaluating the filter predicate: As before, Variant 1 contains the code branching once for both filter conditions, and Variant 2 reorders the predicates using a reorder block. Note that the high-level semantics of the Dynamic Blocks from the last section still apply: It is possible to execute either Variant 1 or Variant 2 and both variations compute the same result. Similarly, the reorder block allows switching Variant 2.1 and 2.2, again without changes to the semantics of the program.

The only change we made to the generated code itself is the special control-flow instructions `dynbr`, which denote the start and end of a variant code fragment. This new instruction is conceptually similar to an unconditional branch but additionally marks the variant's start and end block. E.g., the `dynbr` in line 5 of Figure 4 references the blocks enclosing the alternative block that spans from line 7 to line 44. Due to this design, we can reuse most of Umbra's existing code generation infrastructure. We provide helper functions that automatically emit `dynbr` instructions when starting and closing Dynamic Blocks or switching between the variants. We thus only need to ensure correct semantics for the implementation of dynamic operators, with little syntactic overhead.

Overall, our Dynamic Blocks require few changes to the code base, integrate seamlessly into existing code with a familiar syntax, and introduce almost no additional instructions. We now generate optimized machine code that can be efficiently adapted.

### 3.3 Compiling Dynamic Code

After introducing our novel code generation approach, we now describe how to efficiently translate the Dynamic Blocks to assembly code. We propose two novel compilation techniques that avoid recompilation and minimize the runtime overhead to generate new variants. While the first one is specific to Umbra's infrastructure, we generalize it to work on any language supporting indirect branches, e.g., C with computed gotos (supported by GCC or Clang), at the cost of slightly less optimal code.

Before presenting the compilation techniques, we briefly sketch how Umbra usually compiles query plans to machine code. Like its predecessor HyPer, Umbra uses the LLVM compilation framework to generate machine code [30]. However, LLVM's compilation latency can become a significant performance bottleneck and stall the execution. Umbra, therefore, implements its own x86 instruction generator to translate IR programs to machine code, the Flying Start compiler [22]. Compared to LLVM, the Flying Start compiler achieves very low compilation times at the cost of less optimized code. Simultaneous to processing the first chunks of tuples with code generated by Flying Start, Umbra also compiles the query a second time with LLVM to profit from more expensive optimizations. Once this second compilation pass is finished, we switch to the optimized program using Adaptive Execution [23].

With Umbra's custom Flying Start compiler, we can integrate dynamic blocks directly into the generated assembly code. We emit instructions for all variants in the IR Program and embed them into the executable while keeping track of the dynamic blocks. Figure 5 shows the generated assembly instructions for the IR program from Figure 4. While translating the IR program to machine code, we ensure that the dynamic blocks' semantics still apply, i.e., it is possible to reorder or exchange the machine code fragments.

The Dynamic Blocks blend in well with Umbra's existing compiler infrastructure, and with a few changes, we can lower them to optimized machine code. For the Flying Start compiler, the following three modifications were necessary:

(1) We modified the block placement algorithm to keep all basic blocks of a dynamic block next to each other.
(2) The Flying Start compiler uses an interval-based liveness algorithm to determine how long IR values must be alive. In combination with reorder blocks, the lifetime of IR values must be extended to the end of the dynamic block.
(3) The register allocator also has to consider dynamic blocks. At the end of every variant, we restore the register assignment at the start of the dynamic block.

After this first pass, the generated machine code contains fragments for all variants and dynamic blocks, although we might not need them when executing a specific variation of the program. For instance, Variant Ⓑ of our initial example (cf. Figure 1) uses only code fragments from Variant 2 and switches the reorder block's variants. Hence, we need a second pass that assembles the relevant variants and puts them into the correct order.

```
1   ...
2   .loopTuples:
3     mov rbx, qword ptr [rsp+72]
4     mov r12, qword ptr [rsp+16]
5     mov ebx, dword ptr [rbx+r12*4+3932160]    # %l_commitdate
6
Alternative Block                                              Variant 1
7   .enterAlternativeBlock:
8     mov r13, qword ptr [rsp+72]
9     mov r13d, dword ptr [r13+r12*4+3670016]   # %l_shipdate
10    cmp r13d, ebx
11    setb r13b                                 # %predicate1
12    mov r14, qword ptr [rsp+72]
13    mov r14d, dword ptr [r14+r12*4+4194304]   # %l_receiptdate
14    cmp ebx, r14d
15    setb r15b                                 # %predicate2
16    and r13b, r15b                            # %condition
17    cmp r13b, 1
18    jnz .continueScan
19
20  .nextVariant:                                              Variant 2
Reorder Block                                                  Variant 2.1
21    .enterReorderBlock:
22      mov r12, qword ptr [rsp+72]
23      mov r13, qword ptr [rsp+16]
24      mov r12d, dword ptr [r12+r13*4+3670016]   # %l_shipdate1
25      cmp r12d, ebx                             # %condition1
26      jae .continueScan
27
28    .nextVariant_1:                                          Variant 2.2
29      mov r12, qword ptr [rsp+72]
30      mov r13, qword ptr [rsp+16]
31      mov r12d, dword ptr [r12+r13*4+4194304]   # %l_receiptdate1
32      cmp ebx, r12d                             # %condition2
33      jae .continueScan
34
35  .leaveReorderBlock:
36  .leaveAlternativeBlock:
37    ...
38  .continueScan:
39    ...
```

**Figure 5: Dynamic assembly code by the Flying Start compiler for the filter predicates. We execute either Variant 1 or Variant 2 and the code fragments in Variant 2.2 and Variant 2.2 can be switched.**

```
1   ...
2   .loopTuples:
3     mov rbx, qword ptr [rsp+72]
4     mov r12, qword ptr [rsp+16]
5     mov ebx, dword ptr [rbx+r12*4+3932160]    # %l_commitdate
6
7   .nextVariant:                                              Variant 2
8   .nextVariant_1:                                            Variant 2.2
9     mov r12, qword ptr [rsp+72]
10    mov r13, qword ptr [rsp+16]
11    mov r12d, dword ptr [r12+r13*4+4194304]   # %l_receiptdate1
12    cmp ebx, r12d                             # %condition2
13    jae .continueScan
14
15  .enterReorderBlock:                                        Variant 2.1
16    mov r12, qword ptr [rsp+72]
17    mov r13, qword ptr [rsp+16]
18    mov r12d, dword ptr [r12+r13*4+3670016]   # %l_shipdate1
19    cmp r12d, ebx                             # %condition1
20    jae .continueScan
21
22  .leaveReorderBlock:
23  .leaveAlternativeBlock:
24    ...
25  .continueScan:
26    ...
```

**Figure 6: Assembly code for Variant Ⓑ by copying the code fragments from Figure 5.**

Since we generate dynamic-block-aware assembly, this can be done by copying the translated code fragments to a new buffer and patching any relative jump offsets. Figure 6 shows the machine code for our example assembled with this approach. We cache

```
1  // We initialize dynamicBranches upfront, e.g., Variant (B):
2  // const void* dynamicBranches = {&&nextVariant, &&nextVariant_1,
3  //                                &&leaveReorderBlock, &&enterReorderBlock}
4  ...
5  loopTuples:
6    int32_t l_commitdate = (data + 3932160)[localTid];
7    goto *dynamicBranches[0];
8
Alternative Block                                              Variant 1
9  enterAlternativeBlock:
10   int32_t l_shipdate = (data + 3670016)[localTid];
11   bool predicate1 = l_shipdate < l_commitdate;
12   int32_t l_receiptdate = (data + 4194304)[localTid];
13   bool predicate2 = l_commitdate < l_receiptdate;
14   if(!(predicate1 & predicate2)) goto continueScan;
15   goto leaveAlternativeBlock;
16
17 nextVariant:                                                  Variant 2
18   goto *dynamicBranches[1];
19
Reorder Block                                                  Variant 2.1
20 enterReorderBlock:
21   int32_t l_shipdate1 = (data + 3670016)[localTid];
22   if(!(l_shipdate1 < l_commitdate)) goto continueScan;
23   goto *dynamicBranches[2];
24
25 nextVariant_1:                                                Variant 2.2
26   int32_t l_receiptdate1 = (data + 4194304)[localTid];
27   if(!(l_commitdate < l_receiptdate1)) goto continueScan;
28   goto *dynamicBranches[3];
29
30 leaveReorderBlock:
31 leaveAlternativeBlock:
32   ...
33 continueScan:
34   ...
```

**Figure 7: Adaptive C code for the filter predicates. The `dynamicBranches` array is computed in function's preamble for the given variant.**

the generated programs to reduce L1-instruction cache misses for repeated executions of the same variant.

We now discuss a second compilation technique that integrates dynamic blocks into higher-level programming languages like C or LLVM's intermediate representation. Instead of copying the relevant code fragments into a new executable, we skip fragments or change their execution order using computed gotos. Figure 7 shows adaptive C code generated from the IR in Figure 4: As before, we compile the program with dynamic blocks into one executable, but this time we place a jump instruction before the dynamic block and at the end of each variant. To change the control flow and execute different variations, we now modify the entries in the `dynamicBranches` array.

This technique is similar to NoisePage's Permutable Compiled Queries [29] approach. However, instead of manipulating function pointer arrays, we use arrays of label addresses in the current function. With LLVM's `indirectbr` instruction, we can generate similar code using LLVM as compiler backend[1]. We implemented this technique in the Flying Start backend, but we observe a higher overhead compared to the first approach that produces optimized machine code for every variant. We also considered compiling each variant individually; however, the compilation times increased by more than one order of magnitude, and the quality of the generated code did not improve substantially.

Both compilation techniques for Dynamic Blocks proposed in this section produce highly optimized code and avoid recompilations. While the first approach requires a custom compiler, the second technique can be implemented on top of LLVM/Clang or

[1]https://blog.llvm.org/2010/01/address-of-label-and-indirect-branches.html

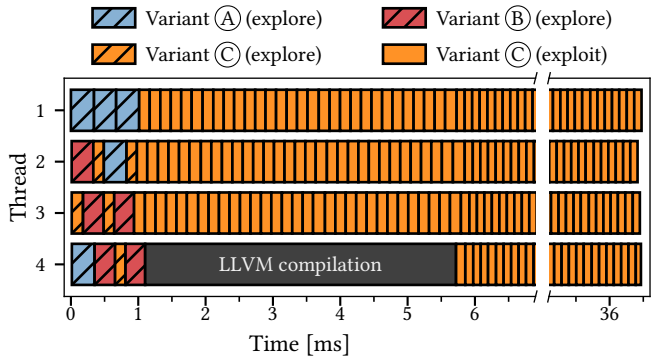



**Figure 8: Execution trace for the example query evaluating all three implementations from Figure 1. After compilation with LLVM is finished, Umbra switches to the faster code and exploits it for the remaining 30 ms.**

GCC. This, however, comes at the cost of a minor performance loss due to the additional jump instructions. The following section presents a new execution strategy for dynamic programs that can hide this overhead almost entirely.

## 4 DYNAMIC EXECUTION

Dynamic Blocks efficiently generate all variants of a query without recompilation. Compiling database systems can use this new technique to try different query implementations and choose the best-performing one. In this section, we propose a dynamic execution strategy that finds the optimal variant after a short exploration phase and combines our Flying Start compiler for Dynamic Blocks with LLVM.

Our dynamic execution strategy is based on Adaptive Execution by Kohn et al. [23] for HyPer. Umbra already uses Adaptive Execution to hide the LLVM's compilation overhead: While compiling the query with LLVM, it processes the first morsels[2] using its Flying Start compiler or a virtual machine. We extend this idea for AQP and use the first few morsels to explore every variation and record the runtime. After evaluating each variant, we choose the best-performing implementation and process the remaining data. Like Adaptive Execution, the dynamic execution strategy compiles the pipeline a second time with LLVM to optimize the machine code even further. However, for the optimized version, we compile the program without any Dynamic Blocks and use only the code fragments from the best-performing implementation.

Figure 8 shows an execution trace for our running example with the dynamic execution strategy. First, Umbra evaluates all three implementations on five morsels and records their runtime during a short exploration phase. We then choose the variant with the lowest average runtime, in this case, Variant (C), and execute it on the remaining morsels. This starts the exploitation phase, and we compile the best-performing implementation with LLVM's optimizations. While compiling the optimized code, we already use Variant (C) but switch to the optimized code after finishing the

[2]Morsels are small chunks of work with a few thousand tuples that HyPer and Umbra use for parallelizing query processing [25].

LLVM compilation. Note that for recompiling the best-performing variation with LLVM, we do not translate the query a second time to the intermediate representation. Instead, we reuse the IR program with dynamic blocks from the first translation pass and collect the relevant code fragment as we do on the assembly level. LLVM then compiles only one implementation and generates highly optimized code without additional jump instructions.

Unlike other systems, our execution strategy performs only one exploration phase and cannot react to changing data distributions. Micro Adaptivity in VectorWise, for example, uses a multi-armed bandit algorithm to find the optimal implementation and regularly re-evaluates the variants [35, 42]. We decided against this approach since recompiling code with LLVM is too expensive, and for most queries, the initial exploration phase already finds the optimal implementation. Nevertheless, the dynamic execution strategy can be extended to change the exploited variant later on and react to skew in the data. However, we propose to monitor the execution times of the currently executed variant and only re-evaluate the variants if its performance changes.

Although our Dynamic Blocks are lightweight and the dynamic execution strategy performs only one exploration phase, there still is an overhead. Adapting every pipeline would introduce a considerable opportunity cost, i.e., evaluating every last variation will take a long time, in which we could already exploit one of the other options. We balance these costs with another optimization pass of our query optimizer that adapts operators only when beneficial and a performance boost is likely. In addition, the adapted operators must process enough morsels to evaluate the different variations and amortize the exploration overhead in the exploration phase.

## 5 APPLICATIONS

After introducing the Dynamic Blocks and how Umbra decides which variant to execute, we now demonstrate their application by implementing two well-known dynamic optimizations in Umbra. Inspired by the adaptive operators in NoisePage [29], we build dynamic operator implementations for evaluating predicates and reordering joins during execution. Furthermore, we discuss several design decisions and some practical problems that come with this new abstraction.

### 5.1 Dynamic Predicates

Our first application for Dynamic Blocks adapts the way Umbra evaluates filter predicates. Our initial example already demonstrated this optimization to motivate AQP and illustrate the code generation process. We consider join $\bowtie$ and select $\sigma$ operators that evaluate conditions that filter tuples. Both operators represent the boolean filter condition as an n-ary conjunctive expression. Each conjunctive component can itself be a complex expression, e.g., a comparison or a nested disjunction. For our adaptive execution, we only consider reordering the topmost conjunction and otherwise consider the expression opaque. To optimize the conjunctive expression, Umbra already applies several heuristics, like eliminating duplicated terms or extracting common ones.

If an operator qualifies for dynamic execution, i.e., it processes enough tuples, we adapt its execution. Using a reorder block, the order in which the terms are evaluated can be changed at runtime (cf. Variant Ⓐ and Variant Ⓑ from Figure 1). Every term of the top-level conjunction is placed in a different variant of the dynamic block, and a conditional branch jumps to the skip block if the term is false. Furthermore, for conjunctions that consist of up to three terms, we generated a second implementation with a single combined branch (all terms are evaluated together as in Variant Ⓒ). We empirically determined three predicates as a heuristic balancing the execution overhead of combined execution and the performance penalty of branch mispredictions.

Implementing this optimization with the Dynamic Blocks from Section 3 is straightforward; there is only one pitfall. We generally want to load columns only once and keep them in registers, but also

```
auto& cdg = getCodeGen();
// One condition, dynamic code not needed.
if (predicates.size() == 1) {
  cdg.branchIfNot(cdg.derive(predicates[0]), skipBlock);
  return;
}

// Find columns needed by multiple terms and load them.
auto columns = findDuplicateColumns(predicates);
for (auto& column : columns)
  cdg.loadColumn(column);

if (predicates.size() <= 3) {
  // Start an alternative block. The generated code will
  // be placed in the first variant of the block.
  cdg.enterAlternativeBlock();

  // Evaluate the predicates without branches.
  CodegenBool result(cdg, true);
  for (auto& predicate : predicates)
    result = cdg.and(result, cdg.derive(predicate));

  // Branch, if the conjunction is false
  cdg.branchIfNot(result, skipBlock);

  // Switch to the second variant in the alternative.
  cdg.nextVariant();
}

// Start the reorder block.
cdg.enterReorderBlock();

// Jump after each term
for (unsigned i = 0; i < predicates.size(); i++) {
  // Place the code in different variants.
  if(i > 0) cdg.nextVariant();

  cdg.branchIfNot(cdg.derive(predicates[i]), skipBlock);
}

// Leave the reorder block. The generated code will be
// placed in the surrounding dynamic block.
cdg.leaveReorderBlock(alternative);

// Leave the alternatve block, if one was used.
if (predicates.size() <= 3) cdg.leaveAlternativeBlock();
return;
```

**Figure 9: C++ code for generating dynamic predicates. We underlined all functions that generate code in Umbra's IR.**

lazily when we first need its data to avoid loading data for filtered tuples. For instance, consider the initial example from Figure 1, where both comparisons need column `l_commitdate`. Usually, we load the column on the first access, i.e., in Variant Ⓐ while evaluating `l_shipdate < l_commitdate`, and then reuse the value in the second term. However, when reordering the two predicates, we are not certain which comparison is first, and we would need to load the column twice. We instead load all columns accessed by more than one term before the dynamic block. The generated code for our initial example in Figure 4 shows this in cf. lines 3 and 4, where we load the commit date before the dynamic blocks begin.

Loading these columns earlier is again a trade-off that can lead to suboptimal code: If the first term skips the remaining predicates, columns accessed only by subsequent terms are loaded unnecessarily. However, the final, optimized code avoids this problem due to our dynamic execution strategy. After choosing the best implementation, LLVM reoptimizes the code and moves the column accesses to the predicate that first uses it.

Figure 9 shows the C++ code that generates the Dynamic Blocks for filter predicates. We first load all columns needed by more than one term in the conjunction (cf. line 9). If the conjunction does not consist of more than three terms, we generate the variant with a combined branch and place it into the first variant of the alternative block. Note that we only create alternatives when needed: line 16 allocates a new dynamic block, and in line 27, we switch to the second variant for the implementation with separate branches. Afterward, we generate the code that branches after each term: Every variant of the reorder block evaluates one term and immediately jumps to the skip block to process the next tuple if the statement is false. Finally, we close the reorder block in line 43, and if a combined implementation was generated, we leave the alternative block as well (cf. line 46).

The Dynamic Blocks generate efficient code for evaluating the predicates and allow adaptive query processing. During execution, we test each variant and find the best implementation for the hardware we are running on.

## 5.2 Dynamic Join Probes

The second optimization we implemented in Umbra dynamically reorders join probes. Adaptive join reordering is not new and has been used in several systems before [3, 28, 29]. We extend this idea by combining it with predicate pullup to reorder potentially expensive filter predicates after or between the joins [9]. Especially in modern main-memory database systems, performing a hash join can be faster than evaluating expensive filter predicates. For example, in a selective join, the probed hash table can be small and fit in the CPU's cache while loading a string column to evaluate, e.g., a `like` condition, might cause a cache miss. Ideally, the join is also more selective than the predicate and eliminates more tuples, reducing the number of executed instructions, branch misses, and cache misses.

However, finding a good heuristic for this optimization is even more complicated than ordering filter predicates. Several factors are at play: Besides the join and predicate selectivities, the size of the hash tables, memory access patterns, and the join condition are decisive for the operators' performance. In addition, the underlying

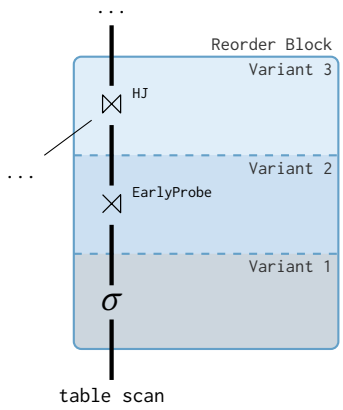



**Figure 10: Reordering select, early probe, and hash join operators using Dynamic Blocks.**

hardware characteristics, like cache miss penalties or cache sizes, are relevant. Estimating all these factors correctly during query optimization is difficult, and a suboptimal execution order reduces the performance significantly. Adaptive query processing solves this problem: We can gather precise performance statistics and choose the optimal plan by exploring the possible implementations at run-time.

Besides hash joins and predicates, we also reorder *early probe* operators ⋊ that perform a semi-join with a bloom filter instead of a hash table [5, 40]. Unlike early probes or selections, joins can produce unmatched tuples or update the entries in the hashtable. We, therefore, consider only inner or right-semi joins for reordering. Furthermore, the reordered joins must not access a column produced by another join, i.e., all join columns must originate from the base table.

Figure 10 shows an adapted query plan that reorders hash joins ⋈, early probes ⋊, and selections $\sigma$. Although the operators perform entirely different operations, we can easily adapt them using Dynamic Blocks. Every operator generates a code fragment for probing the hash table or bloom filter or evaluating the predicates and places it into the reorder block.

The dynamic join probes compose naturally with our dynamic predicates for selections $\sigma$ and we can apply both optimizations to the same operator. Especially in such complex scenarios, Dynamic Blocks provide a way to express and reason about the possible alternatives. Umbra now finds the optimal ordering dynamically during execution, based on the actual runtime and not estimations.

# 6 EVALUATION

In this chapter, we present an experimental evaluation of the Dynamic Blocks in our research RDBMS Umbra [31]. We start with a high-level analysis of the performance impact of AQP on four well-known OLAP benchmarks: The TPC-H, the TPC-DS, the star schema, and the join order benchmark (JOB) [27]. Next, we perform an in-depth evaluation of the implemented dynamic optimizations and discuss some of the queries from TPC-H with high performance gains or losses.

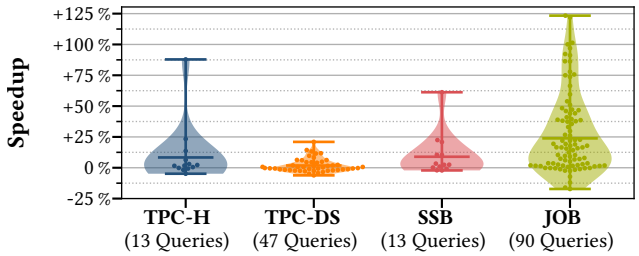



**Figure 11: Speedup of Umbra$^{AQP}$ over to Umbra without AQP for the adapted queries.**

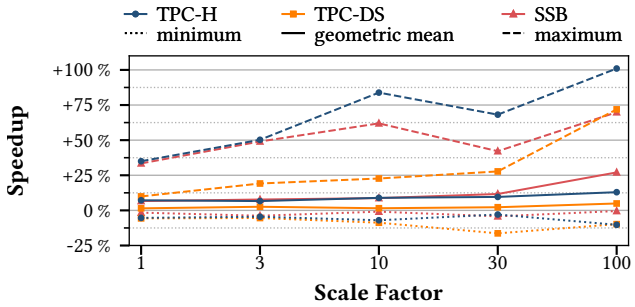



**Figure 12: Speedup of Umbra$^{AQP}$ relative to Umbra for increasing scale factors. We report the minimum, maximum, and geometric mean of all adapted queries per benchmark.**

## 6.1 Experimental Setup

We ran all experiments on an Intel Xeon Gold 6338 CPU (Icelake, 2.0 GHz- 3.2 GHz) with 32 cores and 256 GB of memory. The machine has Ubuntu 21.10 (Kernel 5.13) installed and uses gcc 11.2 and llvm 13.0 for compilation. We measure the performance on 32 hardware threads and report the average runtime of 10 repetitions. Unless stated otherwise, the experiments are conducted on the TPC-H, TPC-DS, and SSB datasets at scale factor 10.

We consider only queries that use at least one of the dynamic optimization. For the given configuration, about 65 % of the queries (163 of 251) from the four benchmarks qualify, and Umbra executes them adaptively. However, this number varies for different scale factors and execution threads, since there might not be enough morsel available to explore the variations and exploit them afterward. We evaluate every variant five times and choose the best-performing implementation for exploitation based on the runtime measurements. In our experiments, five repetitions per variant yielded the best tradeoff between the length of the exploration phase and the probability of selecting the optimal implementation.

In the experiments, we compare the static version of Umbra (**Umbra**) against the adaptive version **Umbra$^{AQP}$** with our two dynamic optimizations and the custom compiler for the Dynamic Blocks. When evaluating one of the optimizations individually, we denote them as **Umbra$^{Pred}$** (cf., Section 5.1) and **Umbra$^{Join}$** (cf., Section 5.2).

## 6.2 End-To-End Benchmarks

We begin by analyzing the overall speedup that we achieve with the two adaptive optimizations combined with Dynamic Blocks. Figure 11 shows the result for the four benchmarks: We use the static version of Umbra without AQP as baseline and compute the speedup based on the total execution time, including optimizing and compiling the queries. Overall, **Umbra$^{AQP}$** is in all four benchmarks faster than **Umbra**. Especially for the join order benchmark, we observe significant performance improvements, as it involves more complex join constellations and unpredictable string predicates. Speedups of more than 100 % are possible and on average, adapting the queries improves the runtime by 25 % in 90 of 114 queries.

For roughly half of the queries, we observe no or only minor performance improvements (< 5 %) as **Umbra** already chooses the optimal or a near-optimal implementation. Since AQP has a minor overhead, **Umbra$^{AQP}$** loses performance in 29 of the 163 queries and is slower than **Umbra**. We identified three different causes for the performance losses:

(1) Exploring the different variations can be expensive, and evaluating a slow variant increases the overall runtime,
(2) The dynamic execution strategy does not always find the optimal implementation and chooses a variant that is slightly slower than the default implementation, and
(3) Our modified Flying Start compiler for translating Dynamic Blocks to machine code generates slightly different programs that can be slower (we evaluate the impact of Dynamic Blocks on the execution performance in more detail in Section 6.4).

However, the losses are mostly marginal, and the runtime increases by more than 5 % in only five queries.

In order to analyze the effects of AQP on smaller and larger datasets as well, we repeated the experiment for different scale factors of TPC-H, TPC-DS, and SSB. Figure 12 shows the minimum, the maximum, and the mean speedup of **Umbra$^{AQP}$** relative to **Umbra**. The maximum speedup grows with the dataset size as we spend more time processing adapted pipelines and exploiting the optimal variation. At scale factor 30, the performance gains drop again as the intermediate structures (mostly hash tables) outgrow the CPU's L3 cache, and query processing becomes more expensive. Nevertheless, the overall performance improvements remain almost constant for different scale factors as most adapted queries do not benefit much from the optimizations.

To summarize, Adaptive Query Processing with Dynamic Blocks achieves stable performance and the dynamic execution strategy consistently finds a good implementation to execute. While the maximum speedup improves for larger datasets, slowdowns do not increase, and at scale factor 100, no query loses more than 10 %.

## 6.3 In-Depth Analysis

In this section, we evaluate the two dynamic optimizations from Section 5 and analyze their impact on the runtime. We inspect five queries from TPC-H at scale factor 100, where AQP improves or degrades the performance significantly. Figure 13 shows the runtime of the queries for the different versions of Umbra: While *Q2* and *Q19* achieve up to 2× speedup with Adaptive Query Processing, the execution time in *Q6* and *Q12* improves only by 14 % and 24 %,

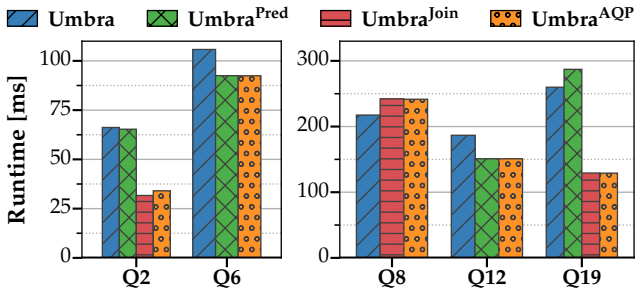



**Figure 13: Runtime comparison of the adaptive Umbra versions for the queries with the highest speedup/slowdown from TPC-H SF 100.**

respectivily. In *Q8*, we even observe a slowdown of 10 % compared to the Umbra version without AQP. However, *Q8* is the only query in TPC-H with a significant performance loss.

*Q2* — This query can apply both dynamic optimizations: It adapts the filter conditions on the `part` table and reorders two joins after scanning the `partsupp` table. Adapting the predicates does not improve the runtime since **Umbra** estimates the selectivities correctly, and the second predicate is an expensive `like` condition. Changing the evaluation order or executing the predicates together does not pay off, and **Umbra$^{Pred}$** evaluates the same plan.

The Dynamic Joins optimization, in contrast, improves the runtime by roughly 100 %. Umbra's optimizer selects a suboptimal join order, and joining `partsupp` with the filtered `part` table first, cuts the execution time in the pipeline in half. **Umbra$^{Join}$** and **Umbra$^{AQP}$** explore the two possible join orders in the pipeline and choose the faster one for the exploitation phase, resulting in a significant performance gain for the query.

*Q6 & Q12* — The Dynamic Predicates optimization adapts in both queries the implementation of the filter predicates on the `lineitem` table. *Q6* evaluates two `between` statements and one comparison on numeric/date columns; hence, executing the predicates requires only a few instructions and is very cheap. Umbra correctly estimates the selectivities and finds the optimal execution order with a separate branch implementation. However, a combined branch implementation, like Variant Ⓒ from the initial example, performs better as it reduces the number of branch misses by 10×. As a result, **Umbra$^{Pred}$** exploits the implementation without separate branches and is 14 % faster.

In *Q12*, we adapt the four predicates and, therefore, generate only dynamic code for reordering the code fragments with separate branches. As before, Umbra's optimizer finds the optimal order according to the predicate selectivities. However, it does not consider the execution costs of the individual predicates and executes an expensive `in` statement on a string column second. With AQP, we move this predicate to the end and evaluate two cheap comparisons on date columns before, although they are less restrictive. **Umbra$^{AQP}$** is 35 ms faster than **Umbra**, which runs the query in 185 ms.

*Q8* — This query is one of the few queries where Umbra with AQP is slower than the default Umbra version. **Umbra$^{AQP}$** uses the Dynamic Join optimizations in combination with predicate pull-up to reorder a join and select operator. The reordered variant where the join is executed before the selection is faster, and our dynamic execution strategy decides to exploit this implementation. However, after compiling the code with LLVM, this changes: the implementation used by **Umbra** is now faster than the reordered variant. We cannot adapt the code anymore and have to execute the slightly slower variant, resulting in a 10 % performance loss.

The performance loss is not directly caused by our Adaptive Query Processing technique but instead by the two-stage compilation process in Umbra. The Flying Start compiler for Dynamic Blocks and LLVM generate different programs, and the runtime measurements from the exploration phase are not totally accurate. However, this situation rarely occurs and the optimal implementation from the Flying Start compiler is usually also faster with LLVM. We could solve this problem by adapting the code generated by LLVM as well, but we decided against this as it will slow down other queries.

*Q19* — The last query achieves the highest speedup in TPC-H: Reordering a join and select operator on the `lineitem` table doubles the performance. The join is 45× more selective than the predicates from selection, and executing it first is the better choice.

Adapting the select operator is also possible, but here we observe a performance loss of 9 %. **Umbra$^{Pred}$** chooses a slower variant and exploits it instead of the faster default implementation. In combination with reordering the join and select operator, however, the implementation of the filter predicates does not matter anymore, as the join already eliminates most tuples. We, therefore, observe the same speedup in **Umbra$^{AQP}$** as in **Umbra$^{Join}$**.

Both dynamic optimizations, adapting predicates and reordering joins/selections, significantly improve the performance. Pulling up select operators in the Dynamic Joins optimization achieves the highest performance boost and improves the runtime by more than 2× in TPC-H and JOB. Reordering only joins and early probes is not that effective and fewer queries use this optimization.

With the Dynamic Predicates optimization, high speedups are rare since Umbra's optimizer usually finds the optimal order according to the predicates' selectivities. Nevertheless, for some queries, significant performance gains are possible when more expensive statements are involved, e.g., string comparisons, `in` statements, or `like` predicates. For these queries, executing a less restrictive predicate first is sometimes faster.

## 6.4 Execution Overhead

The Dynamic Blocks offer a low-overhead AQP in compiling database systems. In our final experiment, we inspect their impact on the execution and evaluate their runtime overhead. For generating native machine code with Dynamic Blocks semantics, we modified Umbra's custom Flying Start compiler. These modifications can lead to different block placement and register assignments (cf. Section 3.3). In addition, we have to load some columns earlier for reordering expressions and relational operators.

We use the following experimental setup to measure the overhead: For every adapted query in the four benchmarks, we compare the original runtime without adaptivity against the dynamic code. We deactivated the exploration phase and only executed Umbra's

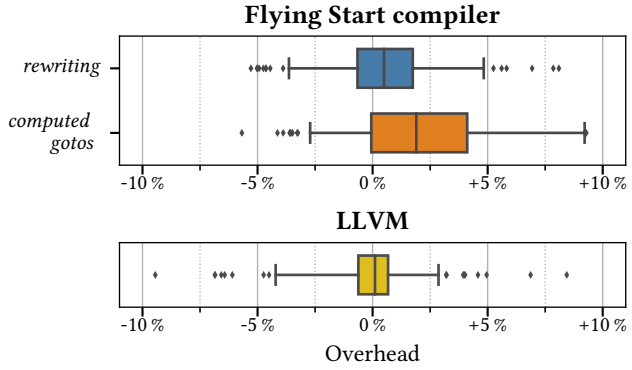



**Figure 14: Execution overhead of the Dynamic Blocks. We compare the dynamic code generated by Umbra$^{AQP}$ to the code without Dynamic Blocks. Umbra$^{AQP}$ executes the default variant, i.e., it does not reorder or adapt the code but still compiles the Dynamic Blocks. We benchmark both compilation techniques for the Flying Start compiler: the approach for assembling new variants from pre-compiled code fragments (*rewriting*) and the implementation with *computed gotos*. For LLVM, we measure the overhead of the optimal implementation generated by Umbra$^{AQP}$ using IR with Dynamic Blocks compared to the optimal variant compiled by Umbra.**

default variant to determine the runtime overhead. Figure 14 shows the results for Umbra's Flying Start compiler and LLVM. For the Flying Start compiler, we consider both compilation techniques. *Rewriting* is our optimized implementation that assembles machine code for every variant. *Computed gotos*, in contrast, assemble the code only once and add indirect jumps to change the control flow during execution. This second technique can also be used with LLVM or C(++) to generate dynamic code.

For most queries, the runtime does not change significantly and reduces or increases by less than 2 %. In some cases, the execution even improves since the dynamic code spills register earlier to the stack, resulting in isolated performance improvements. However, we also see outliers in the other direction, with more than a 5 % loss. Only the computed gotos have a higher execution overhead, but most queries still do not lose more than 5 % performance.

The median runtime overhead is for all compilation techniques less than 2 %, and outliers are rare in the 163 queries we considered. We conclude that the runtime overhead is negligible, and it is more important to find the optimal variant to exploit reliably and fast. Dynamic Blocks allow for fast adaptive execution with Umbra's custom compiler, and after finding the best-performing variant, we remove any overhead using LLVM. Our results also show that Dynamic Blocks can be combined with high-level programming languages using computed gotos. Execution is slightly slower, but recompiling the optimal variant can hide the runtime overhead for long-running queries.

# 7 FUTURE WORK

Our initial motivation for the Dynamic Blocks was to support routing-based Adaptive Query Processing in compiling database systems like Umbra or HyPer. The proposed dynamic optimizations already show a performance boost in Umbra, and we plan to implement more optimizations to improve performance further. Armenatzoglou et al., for instance, adaptively enable or disable early probes [2]. Besides AQP, Dynamic Blocks can be used to efficiently instrument generated code without recompilation. We can sporadically sample the processed data by inserting optional blocks with code fragments to collect runtime statistics into the generated code. These fragments are enabled only for a few morsels during execution to minimize the execution overhead.

Our dynamic execution strategy evaluates all possible implementations in the exploration phase. This approach works fine for our two dynamic optimizations as we rarely generate more than 25 variants to explore per pipeline. However, more dynamic optimizations also produce more implementations that need to be evaluated, and exploring the variant will dominate the overall runtime. A more sophisticated strategy that prunes the search space and eliminates suboptimal implementation early on can solve this problem and efficiently adapts pipelines with more than 100 different variants.

Dynamic Blocks could also reduce compilation times in compiling systems that rely on caches for the generated code. Umbra provides an extensive infrastructure to reduce compilation times (custom intermediate representation, tight integration with LLVM, and a flying start compiler) [22]. Other compiling database systems, like Amazon Redshift [2] or Starling [33], generate C++ code at the cost of high compilation latencies. Redshift mitigates this problem using a distributed code compilation cache. After optimizing the query, it first checks the cache to see if the query plan was compiled before and tries to reuse the cached object files. However, this approach has a major downside: A pipeline with $n$ joins can produce up to $n!$ different join orders. Compiling and caching all these variations is expensive and ineffective. Dynamic Blocks combine all possible join orders into a single executable, and every query later chooses the execution order of the joins independently. As a result, the number of cached executables can be reduced significantly, and the probability of a hit in the code cache increases.

# 8 CONCLUSION

In this paper, we presented an efficient implementation of adaptive query processing in compiling database systems. Our Dynamic Block code generation framework allows developers to embed code fragments into alternative, optional, and reorder blocks and modify the generated code later without recompilation. During query execution, we first explore different variations of the code and then exploit the best-performing implementation. Furthermore, we demonstrated how to generate optimized machine code for Dynamic Blocks and implement adaptive optimizations similar to previous work in vectorized query engines.

We integrated two dynamic optimizations based on Dynamic Blocks that adapt the query plan while executing it into the compiling database system Umbra. Dynamic predicates optimize the execution of filter predicates in table scans, and our second optimization reorders select, join, and early probe operators in pipelines.

When beneficial, AQP improves the performance by more than 2×, while our dynamic execution strategy and custom compiler for Dynamic Blocks avoid slowdowns otherwise. In summary, Dynamic Blocks provide a generic framework to adapt data-centric code and automatically discover the optimal implementation during execution in data processing applications.

## REFERENCES


[1] 2021. *NoisePage - Database Management System Project.* https://noise.page/

[2] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, TJ Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift re-invented. In *SIGMOD/PODS 2022*. https://www.amazon.science/publications/amazon-redshift-re-invented

[3] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD Conference*. ACM, 261–272.

[4] Shivnath Babu and Pedro Bizarro. 2005. Adaptive Query Processing in the Looking Glass. In *CIDR*. www.cidrdb.org, 238–249.

[5] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD Conference*. ACM, 168–180.

[6] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. 2009. Progressive Parametric Query Optimization. *IEEE Trans. Knowl. Data Eng.* 21, 4 (2009), 582–594.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.

[8] Richard L. Cole and Goetz Graefe. 1994. Optimization of Dynamic Query Evaluation Plans. In *SIGMOD Conference*. ACM Press, 150–160.

[9] Andrew Crotty, Alex Galakatos, and Tim Kraska. 2020. Getting Swole: Generating Access-Aware Code with Predicate Pullups. In *ICDE*. IEEE, 1273–1284.

[10] Amol Deshpande and Joseph M. Hellerstein. 2004. Lifting the Burden of History from Adaptive Query Processing. In *VLDB*. Morgan Kaufmann, 948–959.

[11] Amol Deshpande, Joseph M. Hellerstein, and Vijayshankar Raman. 2006. Adaptive query processing: why, how, when, what next. In *SIGMOD Conference*. ACM, 806–807.

[12] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends Databases* 1, 1 (2007), 1–140.

[13] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (2020), 1206–1220.

[14] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *EDBT*. OpenProceedings.org, 313–324.

[15] Anshuman Dutt and Jayant R. Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In *SIGMOD Conference*. ACM, 1039–1050.

[16] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. IEEE Computer Society, 209–218.

[17] Goetz Graefe and Karen Ward. 1989. Dynamic Query Evaluation Plans. In *SIGMOD Conference*. ACM Press, 358–366.

[18] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A. Shah. 2000. Adaptive Query Processing: Technology in Evolution. *IEEE Data Eng. Bull.* 23, 2 (2000), 7–18.

[19] Axel Hertzschuch, Guido Moerkotte, Wolfgang Lehner, Norman May, Florian Wolf, and Lars Fricke. 2021. Small Selectivities Matter: Lifting the Burden of Empty Samples. In *SIGMOD Conference*. ACM, 697–709.

[20] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD Conference*. ACM Press, 106–117.

[21] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[22] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[23] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*. IEEE Computer Society, 197–208.

[24] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, 75–88.

[25] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[27] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[28] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. 2007. Adaptively Reordering Joins during Query Execution. In *ICDE*. IEEE Computer Society, 26–35.

[29] Prashanth Menon, Amadou Ngom, Todd C. Mowry, Andrew Pavlo, and Lin Ma. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *Proc. VLDB Endow.* 14, 2 (2020), 101–113.

[30] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[31] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[32] Thomas Neumann and César A. Galindo-Legaria. 2013. Taking the Edge off Cardinality Estimation Errors using Incremental Execution. In *BTW (LNI)*, Vol. P-214. GI, 73–92.

[33] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD Conference*. ACM, 131–141.

[34] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. In *ICDE*. IEEE, 1758–1761.

[35] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. 2013. Micro adaptivity in Vectorwise. In *SIGMOD Conference*. ACM, 1231–1242.

[36] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. 2003. Using State Modules for Adaptive Query Processing. In *ICDE*. IEEE Computer Society, 353–364.

[37] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *POPL*. ACM Press, 12–27.

[38] Viktor Rosenfeld, Max Heimel, Christoph Viebig, and Volker Markl. 2015. The Operator Variant Selection Problem on Heterogeneous Hardware. In *ADMS@VLDB*. 1–12.

[39] Kenneth A. Ross. 2002. Conjunctive Selection Conditions in Main Memory. In *PODS*. ACM, 109–120.

[40] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. 2001. Integrating Semi-Join-Reducers into State of the Art Query Processors. In *ICDE*. IEEE Computer Society, 575–584.

[41] Feng Tian and David J. DeWitt. 2003. Tuple Routing Strategies for Distributed Eddies. In *VLDB*. Morgan Kaufmann, 333–344.

[42] Joannès Vermorel and Mehryar Mohri. 2005. Multi-armed Bandit Algorithms and Empirical Evaluation. In *ECML (Lecture Notes in Computer Science)*, Vol. 3720. Springer, 437–448.

[43] Eugene Wong and Karel Youssefi. 1976. Decomposition - A Strategy for Query Processing (Abstract). In *SIGMOD Conference*. ACM, 155.