Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Adaptive Query Execution: Dynamically Rewriting Compiled Queries

Tobias Schmidt

Masterarbeit im Elitestudiengang Software Engineering

Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Adaptive Query Execution: Dynamically Rewriting Compiled Queries

Verfasser:	Tobias Schmidt, B.Sc.
Matrikelnummer:	1654247
Beginn der Arbeit:	08. Juni 2021
Abgabe der Arbeit:	08. Dezember 2021
Erstgutachter:	Prof. Dr. Thomas Neumann
Zweitgutachter:	Prof. Alfons Kemper, Ph.D.
Betreuer:	Philipp Fent, M.Sc.

SOFTWARE ENGINEERING

Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, der 2. Dezember 2021 Tobias Schmidt

Abstract

Nowadays, analytical database systems use highly sophisticated data processing techniques like query compilation or vectorization to evaluate queries on millions of tuples. In such systems, query runtimes depend on multiple factors like the query execution plan, the underlying hardware platform, and the operator implementation. Anticipating all these factors during query optimization is difficult, and finding the optimal implementation for a query is almost impossible and requires highly accurate cost models and estimations.

Adaptive Query Processing overcomes these limitations by interleaving query optimization and query execution phases: An initial exploration phase explores different variations for a query and then chooses the best-performing one to process most of the data. Numerous optimizations are possible, like reordering joins and predicates or switching the operators' implementations while executing them. However, the number of variations grows with the dynamic optimizations, and compiling all implementations is prohibitively expensive. We, therefore, propose a different approach that avoids recompilations entirely. Our Dynamic Blocks technique compiles the query once and embeds code fragments for all variants into the generated code. When a different variation is needed, we extract the relevant blocks from the original executable and assemble a new executable.

Internally our Adaptive Query Processing technique consists of three components: a code generation framework to provide different variations for a query, an optimized compiler for lowering the variants to machine code, and a dynamic execution strategy that chooses the optimal implementation while executing the query. We integrate these three components into the research database system Umbra and implement three dynamic optimizations on top that adapt the query plan. Our results show that Adaptive Query Processing improves execution times in a compiling database system like Umbra by up to 2x.

Contents

List of Figures

List of Tables

1. Introduction

Over the last decades, the field of database systems has seen some groundbreaking changes. Hardware trends like the increase of main-memory sizes and the ever-growing number of execution threads per machine paved the way for the evolution of new data processing techniques. Nowadays, machines with more than 1 TB of memory and some hundred execution threads are available and require optimized systems to exploit all resources. Especially the rise of columnar in-memory database systems has set new standards for the processing speed of OLAP queries. Highly parallel query execution engines based on vectorization or just-in-time query compilation process millions of rows in only a few milliseconds. The disk-based database system Umbra recently demonstrated that this performance can also be achieved on SSDs in combination with large in-memory buffers [44].

Yet, from a high-level point of view, the evaluation of queries in relational database systems has not changed much, and the performed steps date back to System R [52]:

1. The database system first parses the query statement and converts it to an expression in relational algebra.
2. Then, the query optimizer simplifies and optimizes the algebraic expression using techniques like predicate pushdown and query unnesting [46]. An essential part of this step is finding a good join order to avoid sub-optimal query execution plans [37]. Once the logical query plan is found, the optimizer chooses a physical implementation for every relational operator in the plan.
3. In the third step, the code generator converts the physical execution plan to machine code.
4. Finally, the query execution engine evaluates the generated code and returns the computed result.

Query engines with the interpreter model do not generate code and instead directly evaluate the physical plan [18].

Modern compiling database systems like Hyper [29], Umbra [44], or NoisePage [2] use the LLVM compiler [1] for code generation. After optimizing the query, they lower the physical plan to LLVM's intermediate representation and compile it. Compared to interpretation-based systems, compilation increases the query execution speed significantly. The generated programs keep data in CPU registers, avoid virtual function calls, and inline code where possible. Nevertheless, this performance boost has a prize: compiling database systems need sophisticated code generation frameworks, and implementing the physical operators becomes more difficult [32].

Usually, database systems do not change the query execution plan after the optimization step. However, the plan or the chosen operator implementations might not be optimal due to optimization mistakes and coarse cost models. Cardinality misestimations, for example, are a well-known problem in query optimization and can lead to sub-optimal join orders [37]. Different hardware platforms can exhibit diverse performance characteristics; hence, finding the optimal implementation is challenging [51]. Therefore, query optimizers apply heuristics to find robust plans that achieve near-optimal performance for most of the queries.

Adaptive Query Processing can mitigate these problems. The basic idea is to postpone some of the query optimizer's decisions and make them at run-time when accurate statistics and cost information are available. For instance, while executing a query, it is possible to choose a different implementation for evaluating relational operators or expressions and change the execution order of filter predicates and joins. Besides the potential performance improvements, adaptive processing allows the execution engine to react to the underlying hardware platform and changing characteristics in the workload. Adaptive Query Processing relaxes the boundary between steps 2 to 4 from the System R model and interleaves the query optimization and query execution steps. However, especially for compiling query engines, this optimization technique is challenging as changes to the query execution plan or the operators' implementation require recompilations of the generated code.

This thesis proposes a novel query compilation technique that allows to exchange operator implementations and reorder code fragments without recompilation. Using this technique, we can generate variations, i.e., different implementations for a query that compute the same result, at run-time with almost no overhead and with only minor performance penalties. During execution, these variations are evaluated, and the best-performing implementation is determined. To demonstrate the practicality of our technique, we integrated it into the research database system Umbra and developed the *Dynamic Blocks* code generation framework for writing adaptive operator implementations. Furthermore, we implemented three dynamic optimizations and evaluate their impact on the TPC-H, TPC-DS, star schema (SSB), and join order (JOB) benchmarks. Our experiments show that Adaptive Query Processing can improve execution time by up to 2x, although Umbra already achieves class-leading performance.

The rest of the thesis is structured as follows: In Chapters 2 and 3, we explore existing work on Adaptive Query Processing in database systems and derive the main challenges for Umbra. Chapter 4 introduces our Dynamic Blocks framework and describes the implementation of the adaptive operators. We then explain in Chapter 5 how the code is compiled and the different variations are generated. Chapter 6 shows the changes to Umbra's runtime system to execute queries dynamically. In Chapter 7, we evaluate the benefits of Adaptive Query Processing on the four benchmarks and provide detailed results. Finally, after discussing some of the design decisions and future work in Chapter 8, we draw a conclusion in Chapter 9.

2. Related Work

Adaptive Query Processing has been around for quite some time and dates back to the Ingres database system [59]. Its one-variable query processor uses a decomposition algorithm for evaluating queries. This approach omits the query execution plan entirely and decides for each tuple individually how to process it. Ingres uses *intra-query* and *intra-operator* adaptivity as it modifies the execution from tuple to tuple in the currently running query [23]. The terms intra-query and intra-operator adaptivity denote the frequency a system adapts its execution. While systems with intra-operator adaptivity re-optimize the execution of an individual operator, intra-query systems use statistics collected by previous operators to improve the evaluation of subsequent operators.

Inter-query adaptivity uses measurements from previous query executions to optimize future queries. DB2's learning optimizer LEO, for example, records the number of produced tuples per operator to improve cardinality estimations [53]. Similar techniques were proposed in [3, 9, 10]. For this thesis, we focus on techniques that introduce adaptivity on an intra-query and intra-operator basis to adapt queries to the data they process.

Besides the distinction based on the system's adaption frequency, Babu and Bizarro [5] differentiate between three different families of systems (a similar taxonomy can be found in [13]):

1. *Plan-based* systems re-optimize the query execution plan if the observed behavior deviates from the estimations. We further distinguish two styles: (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.

2. *Routing-based* systems decide for every tuple or batch independently how to evaluate it and choose different operator implementations or orderings. This approach avoids re-invocations of the query optimizer, and adaptivity is limited to a single operator or pipeline.

3. Adaptive processing in *continuous query-based* systems is similar to plan-based systems. However, this approach reacts to changing statistics and stream characteristics since the data arrives after optimizing the query.

Although Umbra supports stream processing in the form of continuous views [58], we focus on Adaptive Query Processing for non-continuous queries and explore only the

first two approaches in more detail. Nevertheless, Grulich et al. showed that many concepts from standard query engines are also applicable in stream processing engines [20]. Their compilation-based engine Grizzly collects fine-grained data characteristics through instrumentalization and then uses this information to perform data-specific optimizations.

Plan-based Adaption Graefe and Ward use Parametric Query Optimization to adapt prepared statements to user variables that affect parameters like selectivities or memory usage [19]. The optimal plan for such queries depends on the values provided by the user. Hence, a parametric query is compiled with multiple candidate plans optimal for different points in the parameter space. The authors use a special *choose-plan* operator to embed the different plans into one query execution plan. At run-time, this operator then selects the best candidate and executes it. Cole and Graefe later refined this method with dynamic programming to efficiently construct the candidate plans during query optimization [11]. Hulgeri and Sudarshan developed several algorithms for finding the parametric optimal set of plans, i.e., a set of candidate plans that are optimal for all points in the parameter space [24, 25].

Plan Bouquets adopt this idea and use parametric queries for "Query Processing without Selectivity Estimation" [15]. The selectivities of the base table predicates are used as parameter space, and a small "bouquet" of candidate plans is identified from the set of optimal plans. This bouquet is then iteratively executed until a robust plan is found. A major drawback of this optimization technique is the expensive analysis for constructing the optimal set of plans as the search space grows exponentially with the number of parameters (curse of dimensionality) [14].

Progressive Parametric Query Optimization reduces the optimization costs by progressively identifying the candidate plans in multiple executions of the same query [7]. The candidate plans are cached, and the optimizer is only called if no good plan for the given parameters exists. As a result, progressive optimization amortizes the costs for constructing the parametric optimal set of plans and considers only the relevant part of the parameter space.

Mid-query Re-optimization follows a different approach. Instead of preparing or caching multiple plans for the same query, this technique adapts the query execution plan at run-time. Kabra and DeWitt proposed re-optimization in [27]. They add artificial statistics-collectors operators to the query execution plan that gather data distribution and intermediate result sizes. During query execution, these operators compute new estimates for the query optimizer that re-optimizes the remaining plan. The optimizer is only called again if the initial estimates differ significantly from the gathered statistics to minimize the overhead of re-optimization. Furthermore, statistics-collectors are inserted at key points in the query execution plan with high *inaccuracy potential* and substantial impact on execution time.

Ng et al. apply re-optimization to a distributed query processing engine [47]. Besides query characteristics, they also capture statistics on the execution environment, like machine load or resource availability, and consider both factors for finding the optimal query execution plan and machine assignment. Similarly, the Tukwila system generates a partial query execution plan and, after executing it, continues optimizing the query with precise statistics from the intermediate results [26]. Hence, this approach fully interleaves the planning and execution phases. In addition, Tukwila also incorporates techniques from routing-based systems to achieve intra-operator adaptivity.

Markl et al. introduced the concept of *Progressive Query Optimization* [40]. They insert checkpoint operators into the query execution plan that count the number of tuples produced by the previous operator and compare it to precomputed validity ranges. If the number of tuples is outside of this range, the plan is re-optimized. The validity ranges are chosen such that a re-optimization will produce a different plan, and re-optimization is only triggered if beneficial. Unlike mid-query re-optimization, progressive optimization can interrupt the execution of an operator, and partial intermediate results must be discarded. *Proactive Re-optimization* avoids the loss of intermediate results and minimizes the number of re-optimizations using switchable and robust plans [6]. A robust plan is near to optimal most of the time, and a switchable plan contains a set of candidate plans that are interchangeable without losing incomplete intermediate results.

In the last decade, the effectiveness of *Mid-Query Re-optimization* has been demonstrated in several systems. For instance, Neumann and Galindo-Legaria implemented an incremental execution framework in SQL Server [45]. Their approach identifies sub-plans with the potential to change the query execution plan in case of misestimations. At run-time, the sub-plans with the highest uncertainty are executed first, and if the cardinality estimation changes, the plan is re-optimized. Recently, Perron et al. implemented mid-query re-optimization on top of PostgreSQL [48]. They extract sub-queries with high estimations errors into temporary tables and materialize the result before re-optimizing and executing the remainder of the query.

Routing-based Adaption *Tuple-routing* systems insert special, adaptive operators into the query execution plan to perform Adaptive Query Processing. This technique avoids multiple invocations of the query optimizer and delegates the decision of how to process the tuples to the runtime system. Additionally, tuple-routing supports full intra-operator adaptivity, where each tuple can be executed differently (cf. the decomposition algorithm in Ingres [59]).

Avnur and Hellerstein proposed the adaptive *eddy* operator to change the execution order of operators [4]. It discovers the optimal order in which to evaluate the operators while executing them and adaptively re-routes the tuples. As the routing decision is made on a per-tuple basis, the eddy operator can correct misestimation and react to changing data distributions. This approach works well for database operators that can

be pipelined, but it is not suited for blocking operators like aggregating or sorting data.

The eddy operator was successfully adopted in several systems and has been the center of further research for routing-based systems [14]. Tian and DeWitt implemented and evaluated the operator in a distributed stream processing engine [56]. State Modules (SteMs) allow to choose access methods and join algorithms at run-time and increase the adaptivity for joins in combination with the eddy operator [49]. The *STAIR* operator "lifts the burden of history" [12] from the eddy operator by mitigating the effects of routing mistakes. Li et al. follow a similar approach as the eddy operator and reorder index nested-loop joins in a pipelined query execution plan [39].

In the last two decades, the execution model of modern high-performance database systems has shifted from Volcano-style iteration [18] with tuple-at-a-time processing to vectorized execution or compilation-based query engines [31]. This paradigm shift has made it more difficult to hide the performance overhead of routing-based adaption. Nevertheless, Răducanu et al. and Menon et al. demonstrated the practicability of tuple-routing in combination with vectorization by reducing the adaption frequency from a per-tuple basis to vectors or blocks with thousands of tuples [51, 42]. Zeuch et al. investigated the possibility of using the CPU's performance monitoring unit for low-overhead statistics collection in a compiling database system [60]. Their prototype records hardware counters like the number of cache misses and branch mispredictions to find the optimal execution order for base table predicates. However, their approach does not consider the compilation overhead, which can be a significant bottleneck in a compiling database system, as shown by Kersten [30].

3. Adaptive Query Processing

The goal of this thesis is the implementation of a generic and adaptive tuple-routing framework in the compiling database system Umbra [44]. Before looking into our solution, we briefly investigate how other state-of-the-art OLAP systems have implemented adaptivity. We consider the three systems Vectorwise [61], HyPer [29], and NoisePage [2], which all use the routing-based approach with intra-query and intra-operator adaption frequency. Based on this analysis, we then derive three challenges that need to be solved to efficiently implement Adaptive Query Processing in a compiling database system with tuple-oriented query execution.

3.1. Micro Adaptivity in Vectorwise

Vectorwise is a columnar relational database management system with block-oriented query processing [61]. It is based on MonetDB's X100 query engine that performs vector-at-a-time processing in Volcano-like execution pipelines. Relational operators process vectors with hundreds of tuples instead of a single tuple and parallelize the execution using SIMD instructions. Complex operations, like hash table lookups or evaluating expressions, are broken down into low-level computations, so-called *primitive functions*, that perform basic computational actions. Each primitive consumes one or more vectors and produces a new vector with the result. The database system has hundreds of such primitive functions to perform various operations on different data types [8].

Răducanu et al. [51] noted that Vectorwise spends more than 90 % of the time processing a query in primitive functions. Therefore, the primitives offer the potential for further performance improvements. For example, a primitive that filters tuples can either be implemented in a branch-free or branching way. While the branch-free version provides stable performance independent of the selectivity of the filter predicate, the cost of the branching version varies. If the predicate has a very high or low selectivity, branching is almost twice as fast as the branch-free version; otherwise, it is significantly slower due to branch mispredictions.

Ideally, a query optimizer would decide which version to use, but in reality, this is not possible due to several reasons. First of all, there is a fine line between speedup and slowdown for this optimization, and even small misestimations can cause performance degradations. Secondly, the best-performing implementation for a filter primitive can

change over time: for the first half of the processed data, the branch-free version might perform better, while for the second half, branching is optimal. Lastly, the effects of the underlying hardware platform on the performance are difficult to predict and depend on various CPU characteristics like the microarchitecture, cache sizes, branch predictor, and hardware prefetcher. All these factors make it impossible to develop a universal cost model that reliably chooses the best implementation.

The authors, therefore, propose a different approach: *Micro Adaptivity* chooses the optimal implementation for a primitive function dynamically while executing it. For each primitive function, Vectorwise compiles multiple different versions and regularly evaluates these variations. Once the best-performing version is found, it is executed for a thousand vectors, and after that, exploration begins again. Vectorwise measures the performance of a variation through the number of cycles spent in the primitive. Since the primitives are called once for every vector, the function call overhead can be neglected. The entire problem of finding the optimal version of a primitive function can be modeled as a multi-armed bandit problem. For this problem, several solutions exist, like the ε-greedy strategy [57] or Thompson sampling [55]. However, the authors preferred their solution over existing algorithms as it reacts better to changing data distributions and performed best in their benchmarks.

Besides the filter predicate improvement, Răducanu et al. implemented additional optimizations for the Micro Adaptivity framework. They compile the primitives with different compilers (gcc, icc, and clang) and use optimization techniques like loop fission and loop unrolling. Although all compilers generate machine code that performs the same computation, the actual implementations differ widely and can have a significant impact on the overall performance. They saw no clear winner and therefore decided to adaptively switch between different compilers. Loop fission and loop unrolling can reduce the data dependencies in the CPU's execution pipeline, but the benefits of this optimization once again are highly unpredictable.

Although Micro Adaptivity does not adapt execution on a per-tuple basis, it can be classified as tuple-routing adaptivity with intra-operator frequency. Unlike previous approaches, Vectorwise implements Adaptive Query Processing deep inside the runtime system and invisible to the query optimizer. This design decision limits the possible optimizations to the low-level primitive functions and their implementation. In particular, join or predicate reordering, as we have seen with the eddy operator [4], is not possible. Nevertheless, Micro Adaptivity improves query performance for the TPC-H benchmark by almost 10 % and makes query processing more robust to data skew.

3.2. Adaptive Execution in HyPer

HyPer uses a different approach than Vectorwise; instead of vector-at-a-time processing, it compiles the query plan to machine code and executes the generated instructions

on the data in a tuple-at-a-time fashion [29]. Unlike other compiling database systems such as Amazon Redshift [21] or Apache Spark [16], HyPer uses the LLVM compilation framework [1] for generating optimized machine code. Neumann proposed data-centric query compilation and compiles query plans to LLVM's intermediate representation [43].

While LLVM generates highly optimized machine code, it has one major drawback: compilation usually takes between 10 ms and 1 s. Especially for queries that touch only a few thousand tuples, long compilation times are problematic. Compiling the query will dominate the overall runtime as query processing finishes after a few milliseconds. This kind of query is not uncommon for administrative tasks that only read small metadata tables. Additionally, compilation happens single-threaded, and in a system with dozens or even hundreds of threads, only one thread is busy while the others are idle. Hence, upfront compilation became a bottleneck in HyPer, and a solution was needed to improve query latencies.

Kohn et al. [33] propose *Adaptive Execution* to solve this problem. They generate machine code only for long-running queries where the compilation pays off. For short queries, a bytecode interpreter for LLVM IR executes the query immediately. Of course, interpretation is significantly slower than executing machine code, but long compilation times that dominate the runtime are avoided. The interpreter also solves the problem of idle threads during the upfront compilation. While the query is compiled with LLVM, HyPer starts processing the first tuples with the bytecode interpreter, and once the compilation finishes, it switches to the faster, generated code.

Furthermore, Adaptive Execution supports two different compilation modes for LLVM. Unoptimized compilation uses fast instruction selection, and no optimization passes to compile the code as fast as possible. Optimized compilation applies multiple LLVM optimization passes, and compiling is roughly one order of magnitude slower than with the unoptimized configuration. HyPer adaptively chooses the compilation mode depending on the estimated runtime and thereby reacts to the size of the processed data. Overall, Adaptive Execution improves end-to-end runtimes for TPC-H at scale factor 0.01 by almost two orders of magnitude and reduces query latencies significantly in HyPer.

Adaptive Execution in Hyper solves a unique problem to compiling database systems. Nevertheless, this technique shares the characteristics of Adaptive Query Processing. As for Vectorwise, we classify this approach as intra-operator adaptivity using tuple-routing, but as before, adaption happens not on a per-tuple basis but for blocks with a few thousand tuples. However, Adaptive Execution is even more limited than Micro Adaptivity. HyPer always executes the same operations but in different optimization stages, and changes to the query execution plan like reordering joins or predicates are not possible. In addition, the Adaptive Execution framework decides at run-time only whether the query should be compiled or not. Once this decision is made, HyPer waits for LLVM to finish compiling and immediately switches to generated machine code.

3.3. Permutable Compiled Queries in NoisePage

Recently, Menon et al. [42] implemented Adaptive Query Processing in the in-memory research database system NoisePage [2]. Their approach provides comparable functionality as the eddy operator [4]. NoisePage combines vectorized execution and compilation using *Relaxed Operator Fusion* [41]: Long execution pipelines are split into smaller fragments that materialize their results in tuple buffers. NoisePage also uses the LLVM compilation framework for compiling the query plans and enables the compiler's auto-vectorization optimizations to generate vectorized code. Like Vectorwise, NoisePage's operators process batches of tuples (unlike HyPer, which evaluates operators for one tuple at a time).

Adaptive Query Processing in a compiling database system faces a new problem: compiling a function multiple times is just too expensive. For example, when changing the execution order of three filter predicates, the query must be compiled six times. Menon et al. introduced *Permutable Compiled Queries* to solve this problem. Instead of inlining the three predicates, NoisePage generates one function per predicate that is called from the query. The order in which the functions are called is not fixed but determined during execution: NoisePage stores pointers to the predicate functions in an array and rearranges them before executing the generated code for a batch of tuples. Reordering the function pointers avoids recompiling the code, and batch-oriented processing hides the function call overhead as an entire batch of tuples is passed to the function.

Besides predicate reordering, NoisePage can also change the order in which the hash tables are probed in a right-deep query plan. This optimization allows adapting the join order between consecutive hash joins at run-time and reacting to join selectivity misestimations. Furthermore, Menon et al. implemented a "hot set" for aggregations that stores a few heavy-hitter group-by keys. When processing a tuple, NoisePage first checks the set for the key. If the key is in the set, the aggregation is performed in-place; otherwise, a larger hash table that contains all group-by keys is used. Adaptive Query Processing decides whether to use the hot set and which keys to store in it at run-time. NoisePage uses its Permutable Compiled Query technique with arrays of function pointers for adaptive joins and aggregations to avoid recompilations.

Menon et al. profile the generated code in constant intervals and choose the optimal execution strategy based on the collected statistics. NoisePage collects both the selectivity and the cost of evaluating each join or predicate. The optimal execution order is computed based on the rank of the predicates/joins. For the adaptive aggregation, NoisePage tracks the number of distinct keys using HyperLogLog counters. If there are only a few distinct keys, the hot set is used.

The Permutable Compiled Query technique provides full support for routing-based Adaptive Query Processing. It is extensible and allows to express query plan changes and low-level optimizations. The results reported by Menon et al. are also promising.

For TPC-H and the star schema benchmark, they observe up to 71 % speedup.

3.4. Challenges for Compiling Database Systems

Like its predecessor HyPer, Umbra is a compiling database system with tuple-at-a-time query processing. It also supports Adaptive Execution to reduce query latencies [32], but full routing-based adaptivity as in Vectorwise or NoisePage is missing. Since Umbra does not use batch-based execution, neither Micro Adaptivity nor the Permutable Compiled Queries technique work out of the box, and a new solution is needed. For Umbra, we identified the following three challenges that need to be solved to implement Adaptive Query Processing efficiently.

1. As for NoisePage, compiling a query multiple times in Umbra is too expensive. Ideally, we compile the query only once and generate different variations. However, Umbra processes one tuple at a time; hence, hiding the function call overhead is not possible, and the Permutable Compiled Queries approach cannot be used.

2. Umbra's Adaptive Query Processing framework must be expressive enough to perform high-level plan changes, and low-level optimizations like switching between branch-free and branching filter implementations and reordering joins.

3. Furthermore, Umbra already achieves unprecedented performance on standard benchmarks due to its highly optimized execution engine and state-of-the-art query optimizer [44]. Adaptive Query Processing should not reduce the quality of the generated code, and the runtime overhead of exploring different variations must be minimized.

To solve these challenges, we propose three extensions to Umbra's code generation and query execution framework. Our Dynamic Blocks code generation framework is generic and flexible while avoiding recompilations and code duplications. In combination with an optimized compiler and a Dynamic Execution strategy, Adaptive Query Processing is also possible in Umbra. We demonstrate the effectiveness of these three components and their performance throughout this thesis using several benchmarks.

4. Variant-Aware Code Generation

In order to implement Adaptive Query Processing in the compiling database system Umbra, we, first of all, need a way to provide different implementations for relational operators during code generation. Using our Dynamic Blocks code generation framework, which we present in this chapter, developers can adapt pipelines by reordering or exchanging the generated code. Furthermore, we discuss the limitations of our framework and its integration into Umbra's intermediate representation. Finally, we explore the application of dynamic blocks in Umbra to adaptively optimize the evaluation of predicates, joins, and aggregations. Throughout this chapter, we use the queries Q2 (cf. Figure 4.1) and Q12 (cf. Figure 4.2) from the TPC-H benchmark for illustration.

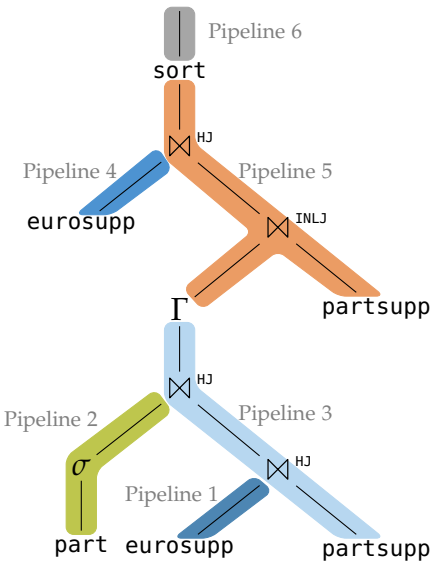
4.1. Recap: Query Compilation in Umbra

Before looking into our new Dynamic Blocks framework, we briefly revisit Umbra's existing *Tidy Tuples* code generation framework [32]. Like its predecessor HyPer [29], Umbra uses the *produce/consume* paradigm proposed by Neumann [43] for data-centric compilation. The model produces highly efficient code by generating tight loops and minimizing the materialization points. Unlike the iterator model [18] or the vectorization model [8], where each operator recursively pulls the next tuple or vector of tuples from its input, in Umbra, each operator pushes the tuples into its parent.

The produce/consume model pushes tuples up the query tree until the next pipeline breaker. An operator breaks the pipeline if it materializes the incoming tuples in memory, e.g., when sorting the tuples, building a hashtable for a join, or aggregating values. In Umbra, a pipeline consists of an initial scan over the incoming tuples (either a base table or another pipeline breaker), the non-breaking operators in between, and the final pipeline breaker. Figure 4.2b shows the query plans and pipelines for TPC-H Q12. The query has three pipeline breakers that materialize tuples: the build side of the hash join to construct the hash table, the group by operator that aggregates all incoming tuples, and the final sort operation. Naturally, this results in four pipelines that are executed sequentially. First, we scan the `lineitem` table and build the hash table for the join (green pipeline). The blue pipeline then iterates over the `orders` table and probes the hash table. Tuples with a join partner are inserted into a second table for the aggregation. In the last two pipelines, Umbra sorts the remaining values and returns them to the user.

```sql
with eurosupp as (
  select s.*
  from supplier s, nation, region
  where s.s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'EUROPE')
select s_acctbal, s_name, n_name,
  p_partkey, p_mfgr, s_address,
  s_phone, s_comment
from part, eurosupp, partsupp,
where p_partkey = ps_partkey
  and s_suppkey = ps_suppkey
  and p_size = 15
  and p_type like '%BRASS'
  and ps_supplycost = (
       select min(ps_supplycost)
       from partsupp, eurosupp
       where p_partkey = ps_partkey
         and s_suppkey = ps_suppkey)
order by s_acctbal desc, n_name,
       s_name, p_partkey
limit 100;
```

(a) SQL code for the query

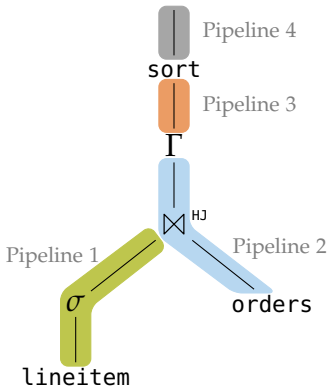


(b) Umbra's query execution plan

Figure 4.1.: TPC-H Q2 (we extracted the common subtree expression `eurosupp`)

```sql
select l_shipmode,
  sum(...) as high_line_count,
  sum(...) as low_line_count
from orders, lineitem
where o_orderkey = l_orderkey
  and l_shipmode in ('MAIL', 'SHIP')
  and l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
  and l_receiptdate between '1994-01-01'
                and '1995-01-01'
group by l_shipmode
order by l_shipmode;
```

(a) SQL code for the query



(b) Umbra's query execution plan

Figure 4.2.: TPC-H Q12

For every pipeline, Umbra generates a function in its custom intermediate representation (IR) and later compiles them to machine code. Umbra's IR is strongly influenced by LLVM's intermediate representation [35] and can be described as a subset with database-specific extensions. Several aspects are identical, including the use of the static single assignment (SSA) form [50] and dividing the program into basic blocks based on terminating instructions (e.g., branches or returns). In the SSA form, every value in the IR program is assigned a value at its definition, and changing a value after creation is not possible; instead, a new value must be defined. The ϕ nodes allow expressing complex data flow in the program. When placed at the beginning of a basic block, a ϕ instruction can select a value depending on the last executed basic block. We need this instruction to implement control-flow statements like `for` loops in the SSA form. Especially the SSA form and ϕ nodes simplify lifetime analysis and facilitate optimizations such as constant propagation or dead code elimination.

Umbra generates instructions according to the produce/consume paradigm. The Tidy Tuples framework provides a translator for every physical operator that implements the consume function. The compilation starts at the bottom of a pipeline (the scan of the base table/pipeline breaker), generates code for iterating the incoming tuples, and passes a virtual representation of the produced tuple to the translator of the next operator. This representation contains all necessary information to access the tuple's columns and allows the translator to compile the next operation. Umbra recursively compiles the pipeline up to the breaking operator, which instead of pushing the tuples to the next operator, materializes them. Figure 4.3a illustrates this: the table scan on `lineitem` generates the `for` loop and `if` statements in lines 1 to 10 and 13. The hash join translator then produces the code for building the hash table (line 12). Depending on the side of the incoming tuple, the translator provides different implementations for consuming the tuples.

After compiling the query plan to the intermediate representation, Umbra lowers the generated pipelines to machine code using one of its backends. Each pipeline compiles to a function that is later called during query execution. On x86-64 machines, Umbra usually employs two backends: the Flying Start compiler directly emits assembly instructions, and an LLVM-based backend translates Umbra IR to LLVM IR and then compiles it using LLVM [32]. While the first backend achieves very low compilation times at the cost of less optimized code, the LLVM-based approach uses multiple optimization passes to produce fast programs. These two backends are combined using Adaptive Execution [33, 34], which first compiles all functions using the Flying Start compiler and while processing the first tuples employs LLVM to generate optimized code for the long-running pipelines.

Morsel-driven execution [36] is used to parallelize the evaluation of pipelines. Besides the code for the pipelines, Umbra also generates functions for initializing, merging, and cleaning up each pipeline. However, since these auxiliary functions are called only a few times and have only a minor impact on the overall performance, we focus on optimizing

```
1  for l in lineitem:
2    if not l_receiptdate between
3         '1994-01-01' and '1995-01-01':
4      continue
5    if not l_shipmode in ('MAIL', 'SHIP'):
6      continue
7    if not l_commitdate < l_receiptdate:
8      continue
9    if not l_shipdate < l_commitdate:
10     continue
11
12   hashtable.put(l)
13 done
```

TableScan Translator

HashJoin Translator

TableScan Translator

(a) Variation produced by Umbra

```
for l in lineitem:
  if not l_receiptdate between
       '1994-01-01' and '1995-01-01':
    continue
  if not l_commitdate < l_receiptdate:
    continue
  if not l_shipdate < l_commitdate:
    continue
  if not l_shipmode in ('MAIL', 'SHIP'):
    continue

  hashtable.put(l)
done
```

(b) Variation with optimal predicate ordering

```
for l in lineitem:
  if not (
      l_shipmode in ('MAIL', 'SHIP')
      and l_commitdate < l_receiptdate
      and l_shipdate < l_commitdate
      and l_receiptdate between
        '1994-01-01' and '1995-01-01'):
    continue

  hashtable.put(l)
done
```

(c) Variation with predication

Figure 4.3.: Different variations for evaluating pipeline 1 of TPC-H Q12 (cf. Figure 4.2).

```
for ps in partsupp:
  for es in eurosupp[ps_suppkey]:
    for p in part[ps_partkey]:
      aggregate(ps ∘ es ∘ p)
    done
  done
done
```

(a) Variation produced by Umbra

```
for ps in partsupp:
  for p in part[ps_partkey]:
    for es in eurosupp[ps_suppkey]:
      aggregate(ps ∘ es ∘ p)
    done
  done
done
```

(b) Variation with optimal join ordering

Figure 4.4.: Different variations for evaluating pipeline 3 of TPC-H Q2 (cf. Figure 4.1).

the pipelines' primary functions.

4.2. Dynamic Blocks Framework

We now extend Umbra's Tidy Tuple framework with our Dynamic Blocks framework for Adaptive Query Processing. Each dynamic block consists of code fragments/variants that can be reordered or exchanged. Doing so produces several different variations of the same pipeline. We demonstrate the dynamic blocks' functionality using two examples:

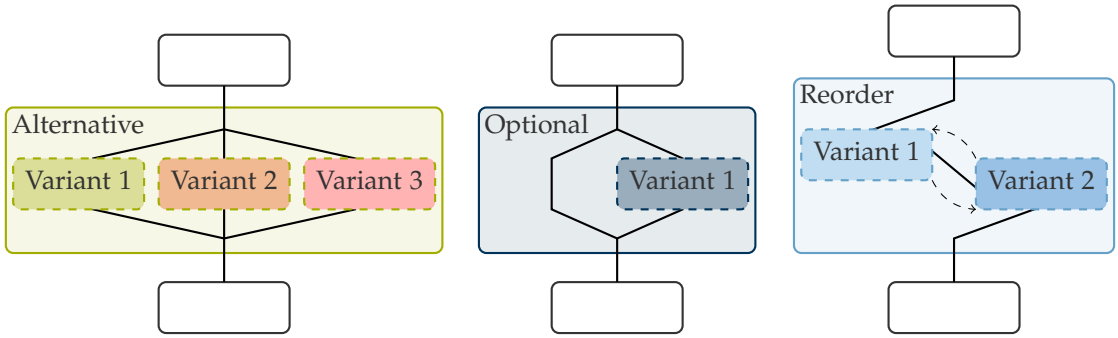
1. For TPC-H Q2, we change the execution order of the two hash joins in the third pipeline (cf. Figure 4.1b). Figure 4.4 shows the possible implementations for the pipeline.

2. For TPC-H Q12, we provide two different variants for evaluating the restrictions on the `lineitem` table: a single-branch implementation that first evaluates the entire condition and then branches, and a multi-branch variant, that evaluates each predicate individually and immediately branches. For the multi-branch implementation, we also adapt the order in which the four predicates are tested.

Especially for Q12, several variations of the original pipeline are possible. One variation with a single branch (cf. Figure 4.3c) and 24 combinations for ordering the four predicates (for instance, Figures 4.3a and 4.3b). However, generating code for all 25 variations is expensive and incurs a substantial overhead during query compilation. Furthermore, the variations share identical code fragments for iterating the `lineitem` table and building the hash table. So instead of generating multiple versions of the same pipeline, we place the code fragments that are exchanged or reordered in dynamic blocks and embed them into a single function generated for the pipeline. Hence, our Dynamic Blocks framework avoids duplication and recompilation during code generation.

We provide three different dynamic block types with different semantics that determine how to construct the pipeline's variations from the contained fragments. We will later use the Flying Start compiler to generate optimized machine code for the dynamic blocks. Figure 4.5 illustrates the semantics of the available block types.

Alternative Blocks The first dynamic block type (cf. Figure 4.5a) chooses between several alternative implementations. For n variants in the alternative block, it is possible to generate n variations of the function. We use this block in Figure 4.6a for Q12 to represent the single-branch variant (green box) and the multi-branch variants (orange box) in the same function.



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

Figure 4.5.: Illustration of the dynamic blocks' effect on the control flow.

Optional Blocks The optional blocks can be used to enable/disable code fragments adaptively. Although it is possible to represent the block as an alternative block with an empty variant, we introduced a new block type that later allows for more optimizations.

Reorder Blocks For changing the evaluation order of joins or predicates, we provide the reorder blocks. As shown in Figure 4.5c, this block type can execute its variants in arbitrary order, resulting in $n!$ variations of the function. Both queries, Q2 and Q12, use the block for changing the code's evaluation order. For Q2, we use the reorder block to swap the two inner `for` loops in Figure 4.6b. Q12 places each predicate into a variant and, thereby, captures all 24 combinations.

The code in Figure 4.6a shows another important feature of our framework: dynamic blocks can be nested arbitrarily deep. In the case of Q12, we reorder the predicates only if the alternative block chooses the second variant. Hence, the combination of the alternative and reorder block produces the 25 variations mentioned above. Furthermore, it is possible to place multiple dynamic blocks in a pipeline that are independent of each other.

For Q2, we need two reorder blocks to change the evaluation order of the join (cf. Figure 4.6b). The first dynamic block swaps the headers of the two inner `for` loops, and the second block reorders the loop tails in lines 5 and 6. However, this implementation can produce incorrect programs. If we reorder only one of the two dynamic blocks, the loop headers and tails do not match anymore. Therefore, we need a mechanism to link dynamic blocks together and allow only valid combinations. When reordering the two loops in our example, we have to move the two code fragments containing the loop header and tail for the first variant inside the second variant. So both the two `for` loops and the `done` statements switch places resulting in Figure 4.6c. In Section 4.4, we

```
1  for l in lineitem:
Alternative Block                                   Variant 1
2    if not (
3         l_shipmode in ('MAIL', 'SHIP')
4         and l_commitdate < l_receiptdate
5         and l_shipdate < l_commitdate
6         and l_receiptdate between
7           '1994-01-01' and '1995-01-01'):
8      continue
                                                    Variant 2
Reorder Block                                       Variant 1
9    if not l_receiptdate between
10        '1994-01-01' and '1995-01-01':
11     continue
                                                    Variant 2
12   if not l_shipmode in ('MAIL', 'SHIP'):
13     continue
                                                    Variant 3
14   if not l_commitdate < l_receiptdate:
15     continue
                                                    Variant 4
16   if not l_shipdate < l_commitdate:
17     continue
18
19   hashtable.put(l)
20 done
```

(a) Pipeline 1 from Q12 with dynamic blocks

```
1  for ps in partsupp:
Reorder Block                                       Variant 1
2    for es in eurosupp[ps_suppkey]:
                                                    Variant 2
3      for p in part[ps_partkey]:
4        aggregate(ps ∘ es ∘ p)
Reorder Block                                       Variant 2
5      done
6    done                                           Variant 1
7  done
```

(b) Pipeline 3 from Q2 with dynamic blocks

```
1  for ps in partsupp:
Reorder Block                                       Variant 2
3    for p in part[ps_partkey]:
                                                    Variant 1
2      for es in eurosupp[ps_suppkey]:
4        aggregate(ps ∘ es ∘ p)
Reorder Block                                       Variant 1
6      done
5    done                                           Variant 2
7  done
```

(c) Pipeline 3 from Q2 with dynamic blocks (reordered)

Figure 4.6.: Pipeline implementations for TPC-H Q2 and Q12 (cf. Figures 4.3 and 4.3) in the Dynamic Blocks framework, unifying different variations in one representation.

describe the integration of our Dynamic Blocks framework and explain how to link dynamic blocks together.

4.3. Semantics and Limitations

After introducing the three dynamics block types, we now have a closer look at their semantics and some limitations when using them. Especially two operations are of interest: reading values assigned in dynamic blocks and how to enter/leave them.

First, we clarify how dynamic blocks are placed in a function. We introduce dynamic blocks while generating the intermediate representation using our code generation framework. Apart from appending new instructions to the current function, we offer the functionality to start a new dynamic block, switch to another variant in this block and close it again. When starting a new dynamic block or variant, all following instructions are inserted into the current variant of this block. At this point, we either 1) start another dynamic block and place it into the current variant, 2) add a new variant to the current

dynamic block, or 3) close the current dynamic block. When leaving the dynamic block, we return to the variant in which the block was placed and append new instructions there.

In general, it is possible to read any value in the program as long as the value is assigned before reading it according to the control flow. Our dynamic blocks framework does not change this. It is still possible to read all values even if they were defined in a dynamic block. However, there is one exception: the control flow between variants of a dynamic block is not deterministic. Hence, values defined in another variant of the same dynamic block cannot be read. For instance, in Figure 4.6a, the statements in the second variant of the alternative block (lines 9 - 17) cannot access values from the first variant since the two variants are mutually exclusive. Similarly, the variants in the reorder block cannot read from each other as the execution order can change, and there is no guarantee that a value is defined before its use. We can only access values when we know for sure that they have been calculated before according to the control flow.

Outside the alternative block, it is possible to access all values defined before. So after line 18 in the example, we can read the values from the alternative and reorder block. Accessing values from alternative or optional blocks requires some caution as it is unknown at compile-time which variant will be used. Hence, we need a mechanism to choose values from the currently executed variant of a dynamic block. In Section 4.4, we propose a simple solution for this problem using the existing ϕ node mechanism.

Umbra's IR provides conditional and unconditional branch instructions to jump between the basic blocks. Usually, these control flow instructions can target any basic block in the function. However, with dynamic blocks, it is not possible to jump between variants of the same dynamic block. Furthermore, jumps into a dynamic block are also not supported to avoid the problem of accessing a code fragment that is not available at run-time in case of an alternative or optional block. But, jumps from a dynamic block into the surrounding variant are still possible.

We have not answered how to enter a dynamic block and switch between its variants yet. In fact, the limitations we just introduced prohibit any control flow into a dynamic block and between the variants. We, therefore, allow the following exception to this rule: it is possible to enter a dynamic block or variant via an unconditional branch to the first basic block. In addition, the source and destination basic blocks must be adjacent to replace the branch later by a no-op. Similarly, we leave the dynamic block using an unconditional branch from the last variant to the first basic block after the dynamic block.

4.4. Integration into Umbra's IR

The next step is to integrate the dynamic blocks into Umbra's IR. For this, we solve the following two problems: first, how to represent the dynamic blocks in the intermediate

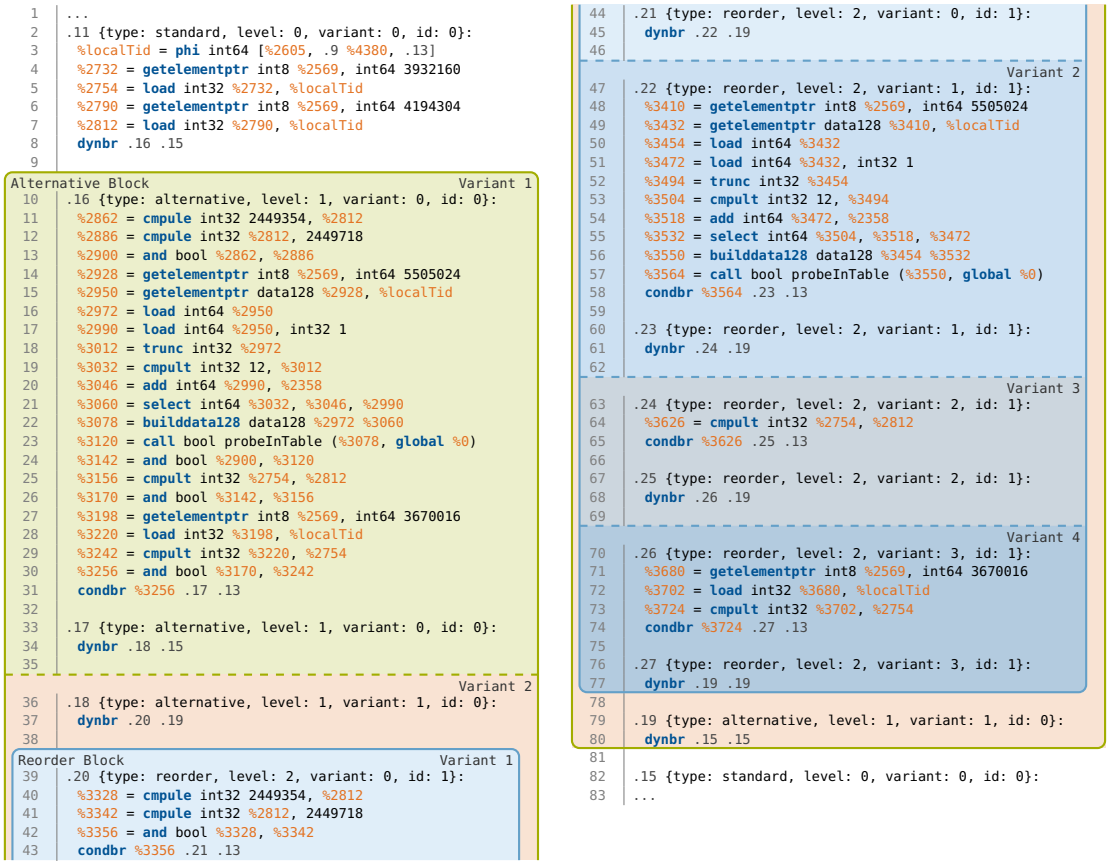


Figure 4.7.: Umbra IR with dynamic blocks for the first pipeline from TPC-H Q12 (cf. Figure 4.6a).

representation, and second, how to pass values between the blocks. Umbra represents the generated functions as a list of basic blocks. Each basic block consists of a list of low-level instructions.

Representation of Dynamic Blocks As for the pseudocode before, we have to assign the instructions to variants of the dynamic blocks. In our intermediate representation, however, all instructions in a basic block are executed together. We, therefore, do not assign single instructions to variants but instead entire basic blocks. With the Dynamic Blocks framework, a function now consists of blocks where each block is either a basic block or a dynamic block. Dynamic blocks contain several variants that are again composed of blocks. This composite pattern results in a tree-like structure when nesting dynamic blocks.

Internally, we maintain three lists for the basic blocks, the variants, and the dynamic blocks. Each basic block has a pointer to the variant it belongs to. This pointer is null if the block is on the top level, i.e., not inside a dynamic block. The variants also have a pointer to their parent dynamic block and the first basic block. Besides the pointers, we give each variant also a number that identifies its position within the dynamic block. For dynamic blocks, we store the type, the nesting level, the number of variants in the block, and a pointer to the first variant. Figure 4.7 shows the Umbra IR for evaluating filter condition in Q12 from Figure 4.6a. We annotate the basic blocks with some of the internal information for illustration: the dynamic block's type (`type`), the dynamic block's nesting level (`level`), the variant number (`variant`), and the id of the dynamic block (`id`).

We provide five helper functions for entering/leaving dynamic blocks and switching between variants.

- `Variant` **`enterDynamicBlock`**`(DynamicBlockType)`
- `Variant` **`leaveDynamicBlock`**`()`
- `void` **`restartDynamicBlock`**`(Variant)`
- `void` **`nextVariant`**`()`
- `void` **`previousVariant`**`()`

These functions automatically start a new basic block, annotate it with the correct values, and take care of the bookkeeping. The first function also generates a new variant and dynamic block and sets the pointers. Both the enter and leave functions return a reference to the first/last variant of the dynamic block, which can later be used to restart it using `restartDynamicBlock` and link variants together. This feature is needed for reordering the joins from TPC-H Q2 in Figure 4.6b. The `nextVariant` function increments the variant number to add a new one, and the `previousVariant` function decrements the number. If the return value from `leaveDynamicBlock` is used for restarting the dynamic block, the variant number is set to the last variant in the dynamic block. The five helper functions make our Dynamic Blocks framework a very lightweight extension of the existing Tidy Tuples framework.

Passing Values between Dynamic Blocks As we already noted in Section 4.3, reading values defined in a variant requires some caution. For alternative and optional blocks, it is unknown at compile-time which variant will be used. Hence, we need a mechanism that reads only values from the executed variant. Luckily Umbra's IR already solves this problem: ϕ nodes select a value based on the last executed basic block.

We use the `phi` instruction to read a value from the used variant. However, to translate the ϕ nodes later, the variants must know the subsequent basic block during execution.

```
1  UInt32 temp = uint32Ptr.load();
2
3  // Start the alternative block                    Variant 1
4  codegen.enterDynamicBlock(alternative);
5  // Get the current basic block and compute
6  auto variant1 = codegen.getCurrentBlock();
7  auto value1 = temp * UInt32(codegen, 2);
8  // Switch to the next variant                     Variant 2
9  codegen.nextVariant(alternative);
10 // Get the current basic block and compute
11 auto variant2 = codegen.getCurrentBlock();
12 auto value2 = temp + temp;
13
14 // Leave the alternative block
15 codegen.leaveDynamicBlock(alternative);
16
17 // Build the PHI node
18 auto ret = codegen.buildRawPhi({{variant1, value1},
19                                 {variant2, value2}});
20 codegen.returnValue(ret);
```

(a) C++ code for generating the example.

```
1  .1 {type: standard, level: 0, variant: 0, id: 0}:
2   %temp = load int32 ptr 0x7ffd70abf7b0
3   dynbr .3 .2
4
Alternative Block                                  Variant 1
5  .3 {type: alternative, level: 1, variant: 0, id: 0}:
6   %value1 = mul int32 %temp, 2
7   dynbr .4 .2
8
                                                   Variant 2
9  .4 {type: alternative, level: 1, variant: 1, id: 0}:
10  %value2 = add int32 %temp, %temp
11  dynbr .2 .2
12
13 .2 {type: standard, level: 0, variant: 0, id: 0}:
14  %ret = phi int32 [%value1, .3 %value2, .4]
15  return %233
```

(b) Umbra IR with dynamic blocks for the example.

Figure 4.8.: Example for reading values from dynamic blocks: we provide two equivalent implementations for computing the expression `(*uint32Ptr * 2)`.

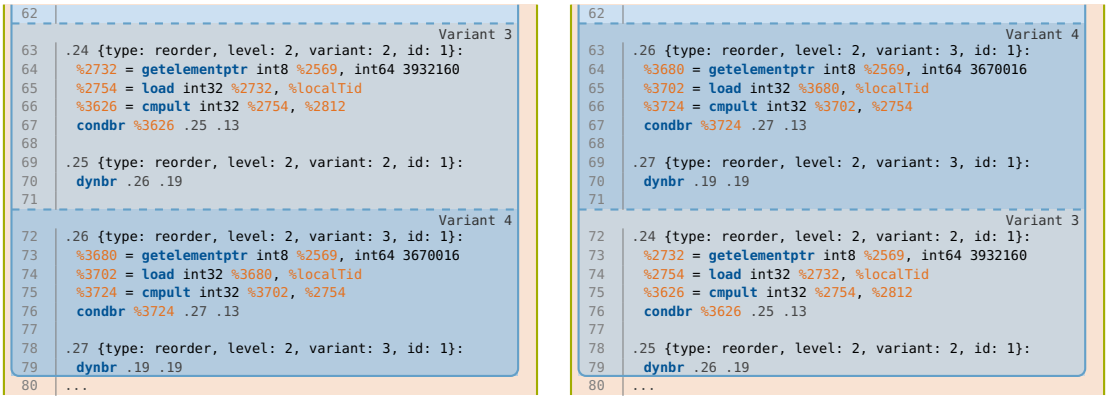
Usually, this is not the next basic block but the first block after the dynamic block. In order to reference this block, we introduce a new `dynbr` instruction that holds pointers to both blocks. Figure 4.8 gives a small example of how these two instructions work together to access values defined in an alternative block. In Figure 4.8a, we provide the corresponding C++ code that generates the intermediate representation for the example. We use the `dynbr` instruction whenever a dynamic block is started or left and when switching the variants. The five helper functions from above automatically place the instruction and reference the first basic block outside the dynamic block.

4.5. Applications and Practical Problems

After introducing our Dynamic Blocks framework, we present three applications of Adaptive Query Processing in Umbra and discuss several design decisions. Inspired by the adaptive operators in NoisePage [42], we also provide dynamic operator implementations for predicate evaluation, join ordering, and aggregation. Furthermore, we describe our new optimizer pass that adds dynamic operators when beneficial.

4.5.1. Dynamic Predicates

In Umbra, three operator types evaluate filter predicates: the select and table scan operators filter their input and joins evaluate a join condition. As we already showed for TPC-H Q12, we can dynamically change the execution order of the clauses in the



(a) default implementation (b) reordered

Figure 4.9.: Invalid Umbra IR with dynamic blocks for the two comparisons using `l_commitdate` from TPC-H Q12 when caching columns.

condition[1].

All three operators represent the condition as a conjunction of terms. A term can either be an actual comparison or, again, a conjunction/disjunction. However, unlike NoisePage, Umbra does not convert the condition into the conjunctive or disjunctive normal form to avoid exponentially growing expressions. Instead, it keeps the original expression tree and applies several heuristics to simplify it, like flattening nested conjunctions and disjunctions, eliminating duplicated terms, or extracting common ones. In order to keep the dynamic implementation as simple as possible and to restrict the number of variations, we decided to reorder only the terms in the top-level conjunction of the condition. As all three operators use the same logic to evaluate the expression, implementing the dynamic predicates requires only minor changes to the existing codebase.

During implementation, we noted a rather fundamental problem with Umbra's caching mechanism: the first time a column is accessed, we cache the loaded value and reuse it in later accesses. For TPC-H Q12, this happens with `l_receiptdate` and `l_commitdate`. The commit date is cached when evaluating the term `l_commitdate < l_receiptdate`, and the second comparison `l_shipdate < l_commitdate` reuses the cached value. Figure 4.9 illustrates this problem: When reordering the two terms, the second comparison in line 66 will try to read a value that is loaded later by the first comparison in line 74. Hence, when caching the columns on the first access, we generate an invalid program that reads from another variant in the same dynamic block.

The same happens for the receipt date: the between statement caches the value, and

[1]Both index-nested loop joins and hash joins evaluate the equi-join condition directly on the index structure, so for these operators, we only consider the remaining residuals for dynamic execution.

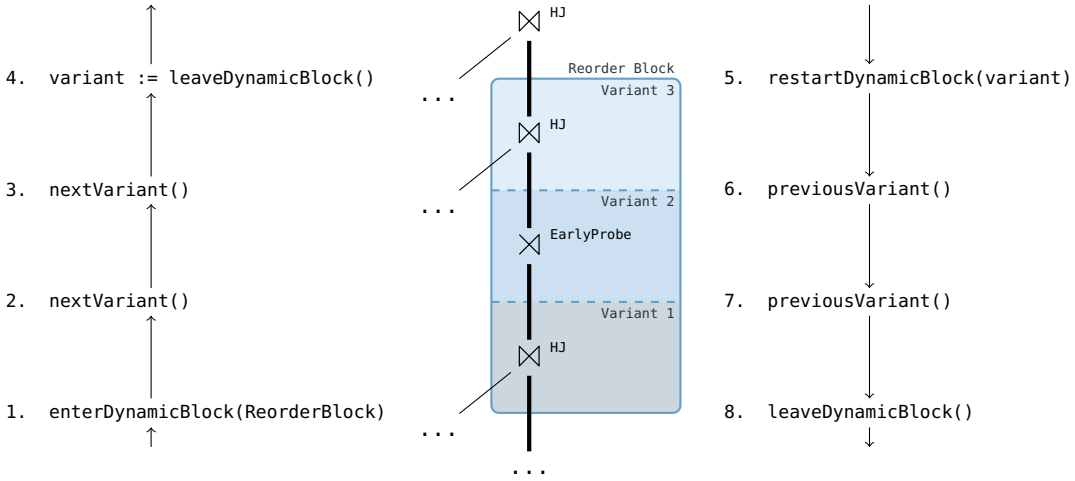


Figure 4.10.: Reordering two hash joins and one early probe using dynamic blocks.

the first comparison reads it. There are two solutions to avoid this problem: either loading the column in every term or caching it in the non-dynamic part of the code. Both loading the same column twice and caching all accessed columns upfront can reduce the performance significantly. We, therefore, decided to cache only columns that are used by more than one term in the conjunction and whereby avoid unnecessary accesses. In lines 4 to 7 of Figure 4.7, we load `l_receiptdate` and `l_commitdate` once and use them later in both variants of the alternative block.

4.5.2. Dynamic Joins

Another optimization used by the NoisePage system is the adaptive reordering of hash joins. In Umbra, we reimplement this feature using our Dynamic Blocks framework, as shown in Figure 4.6b. We can reorder a join under the following conditions:

1. The join must be independent of the previously reordered joins, i.e., it cannot use columns computed by one of the previous joins.
2. The join must not produce unmatched tuples (outer join) or perform additional work (left-semi/-anti join), i.e., we only support reordering inner and right-semi joins.

Besides hash joins, we also support reordering early probes that essentially perform a right-semi join using a register-blocked bloom filter instead of a hash table. Unlike some of the approaches in interpreting systems [4, 39], we cannot change the pipeline's driving table or switch the build side and the probe side. Hence, we can only change the order in which we probe the hash tables and bloom filters.

Figure 4.10 illustrates reordering a chain of two hash joins and one early probe. Since a dynamic join spans multiple operators in the pipeline, reordering the code fragments is more challenging: The first translator in the chain of dynamic joins starts the reorder block. When switching to the next hash join/early probe translator, we start a new variant. The last translator in the chain leaves the reorder block and remembers the last variant. After generating the code fragment for the subsequent operators, we return to the last translator in the chain, restart the reorder block, and finish the hash join. From here on, we return to the previous translator and its variant until we reach the first translator in the chain.

As for the dynamic predicates, we also face the problem of cached columns when reordering joins. Conveniently, we can reuse the same solution as before: columns accessed by more than one join are cached before entering the dynamic block. We also considered index-nested loop joins and eager-right groupjoins for reordering. However, as both join methods tend to process comparatively few tuples, we decided against it.

4.5.3. Dynamic Preaggregation

Umbra uses a thread-local preaggregation phase to optimize the aggregation and exploit morsel-driven parallelism [36]. As shown in Figure 4.11, each thread has a lookup table with 1024 slots and 512 output streams for preaggregating values and partitioning the set of keys. The lookup table stores pointers to the entries in the output stream. When processing a tuple, the aggregation first computes its hash value, retrieves the expected hash value and pointer to the last entry in the output stream from the lookup table, and compares the two hash values. If they are identical, Umbra checks the actual key from the entry in the output stream, and in case of a match, it updates the aggregates; otherwise, a new entry is appended to the output stream, and the lookup table is updated. After adding all tuples to the output streams, Umbra merges the streams and computes the final result. For this step, the operator first shuffles the output streams to process all streams from the same partition on one thread and then inserts the preaggregated values into a global hash table.

Since Umbra's preaggregation lookup table and NoisePage's hot cache have the same effect, we looked for other ways to optimize the aggregation. Although Umbra already uses hardware-optimized hash functions, it is possible to improve this part of the lookup even further. Especially for very few unique keys using the radix as hash can be beneficial and for even fewer keys, we can bypass the hash table entirely and always use the last accessed entry. Figure 4.11 illustrates the two alternative access paths as well.

Besides the standard aggregation (group by), the eager-right groupjoins, and set operations such as union, intersect (all), and except (all) use our dynamic preaggregation.

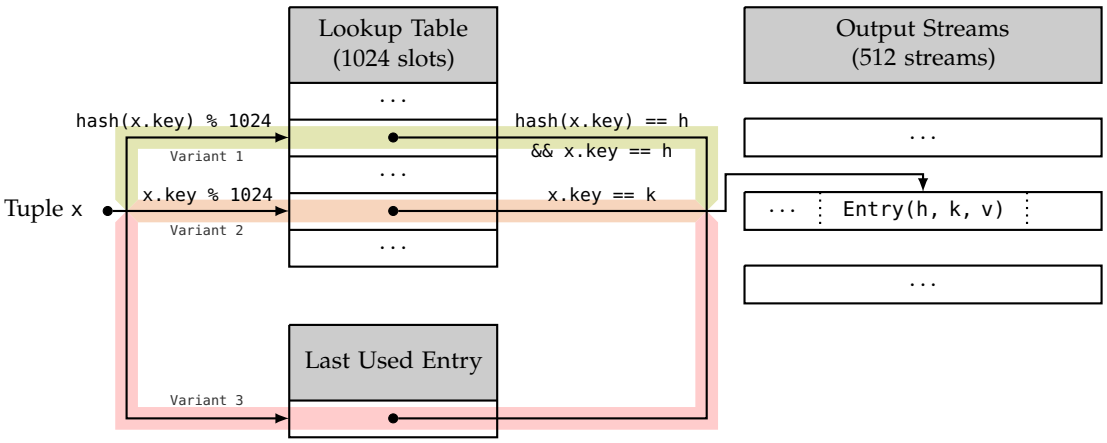


Figure 4.11.: Three alternative access paths for accessing the entries in the output streams during preaggregation. The first variant is the default access path used by Umbra.

4.5.4. Dynamic Optimizer

Although our Dynamic Blocks framework is as lightweight as possible, it still introduces a small overhead during compilation. Hence, using the dynamic operators in every place possible will do more harm than good. So, we added another optimization pass to our query optimizer that introduces dynamic operations only in pipelines where we can expect a performance boost. Umbra uses dynamic operators if the following two conditions are met:

1. The pipeline should process enough tuples to explore the different variations and to find the best-performing one.

2. The operator must have a measurable impact on the pipeline's performance, i.e., it must process a substantial fraction of the incoming tuples.

Algorithm 4.1 sketches the additional optimizer pass. Our optimizer for dynamic operators starts at a pipeline breaker, checks if it produces at least 25 thousand tuples per execution thread, and then traverses the query tree until the next pipeline breaker. For every operator in the pipeline, we check that it processes at least 1 % of the pipeline's incoming tuples and if so, we try to apply our dynamic optimizations. During this pass, we also compute the hash joins and early probes that can be reordered. Since a dynamic join can move every hash join or early probe to the front, we update the estimated number of tuples after the last reordered operator.

Algorithm 4.1: Optimizer for introducing dynamic operators.

```
1  input: Operator inOperator, Double inputSize
2  begin
3    # Consider only pipeline breakers that process enough tuples
4    if not inOperator.isPipelineBreaker() or inputSize < 25000 · degreeOfParallelism():
5      return
6
7    # Minimum number of tuples an operator has to process for dynamic execution
8    minTuples ← inputSize · 0.01
9    # List of hash joins and early probes that are currently reordered
10   dynamicJoins ← []
11
12   op ← inOperator
13   estimatedTuples ← inputSize
14   # Iterate over the pipeline
15   do
16     if estimatedTuples < minTuples:
17       break
18
19     # Check if the current operator is a join and if we can reorder it
20     if (op.isHashJoin() or op.isEarlyProbe()) and canReorderJoin(dynamicJoins, op):
21       dynamicJoins.append(op)
22       op.markAsDynamicJoin()
23     else:
24       dynamicJoins ← []
25       if dynamicJoins.size() == 1:
26         op.child.unmarkDynamicJoin() # Do not reorder a single join
27
28     if op.isTableScan() or op.isSelect() or op.isJoin():
29       # Check if the operator's conjunction has at least 2 terms
30       if op.condition.size() >= 2:
31         op.useDynamicPredicates()
32
33     # Do not update the number of estimated cardinality when reordering joins
34     if dynamicJoins.empty():
35       estimatedTuples ← op.estimateCardinality()
36     op ← op.parent
37   while not op.pipelineBreaker
38
39   if dynamicJoins.size() == 1:
40     op.child.unmarkDynamicJoin() # Do not reorder a single join
41
42   # Check if the pipeline breaker can use dynamic preaggregation
43   if estimatedTuples >= minTuples
44         and (op.isGroupBy() or op.isEagerRightGroupJoin() or op.isSetOperation()):
45     op.useDynamicPreaggregation()
46 end
```

5. Compiling Dynamic Queries

Our Dynamic Blocks framework allows developers to generate multiple variations of the same pipeline with almost no overhead. Still, we have to answer the question of how to generate optimized assembly code for these dynamic queries.

As noted by Menon et al., it is too expensive to generate the code for every variation from scratch [42]. Although our Flying Start compiler reduces compilation times to a few milliseconds, compiling dozens or hundreds of variations will still have a notable impact on the runtime. Instead, we follow the same principle as before for generating the intermediate representation in the Dynamic Blocks framework: compile the pipeline once with all variants and use the generated machine code to assemble different variations later. This approach requires some modification to Umbra's Flying Start compiler, which we describe in Section 5.1, and Section 5.2 shows how to assemble the variations.

5.1. Lowering Umbra IR to Machine Code

Umbra provides several backends for compiling the intermediate representation to machine code. Most notably, the Flying Start compiler introduced by Kersten et al. in [32], which we also use for lowering the dynamic blocks. Apart from its fast compilation time, the backend performs basic block placement, liveness computation, and register allocation in (almost) linear time. After an initial optimization pass that determines the block order and variables' lifetimes, the Flying Start compiler emits assembly instructions in a second pass.

Before discussing how dynamic blocks are compiled, we briefly examine the machine code generated for our example query TPC-H Q12 (cf. Figure 4.6a). Figure 5.1 shows the assembly instructions that our variant-aware version of the Flying Start compiler emits for the first pipeline of the query when reordering the filter predicates. As before, we generate code fragments for every variant, but this time the fragments do not contain high-level IR instructions but low-level machine code. The first variant in the alternative block evaluates all predicates and combines the resulting flags using an `and` instruction (cf. lines 37, 40, and 45). The second variant contains code fragments for each predicate. When examining the code fragments closely, we notice that the variants in the reorder block do not depend on each other and can also be reordered on the assembly level. The same applies to the alternative block, we can run any of the two variants, and the function will still compute the correct result.

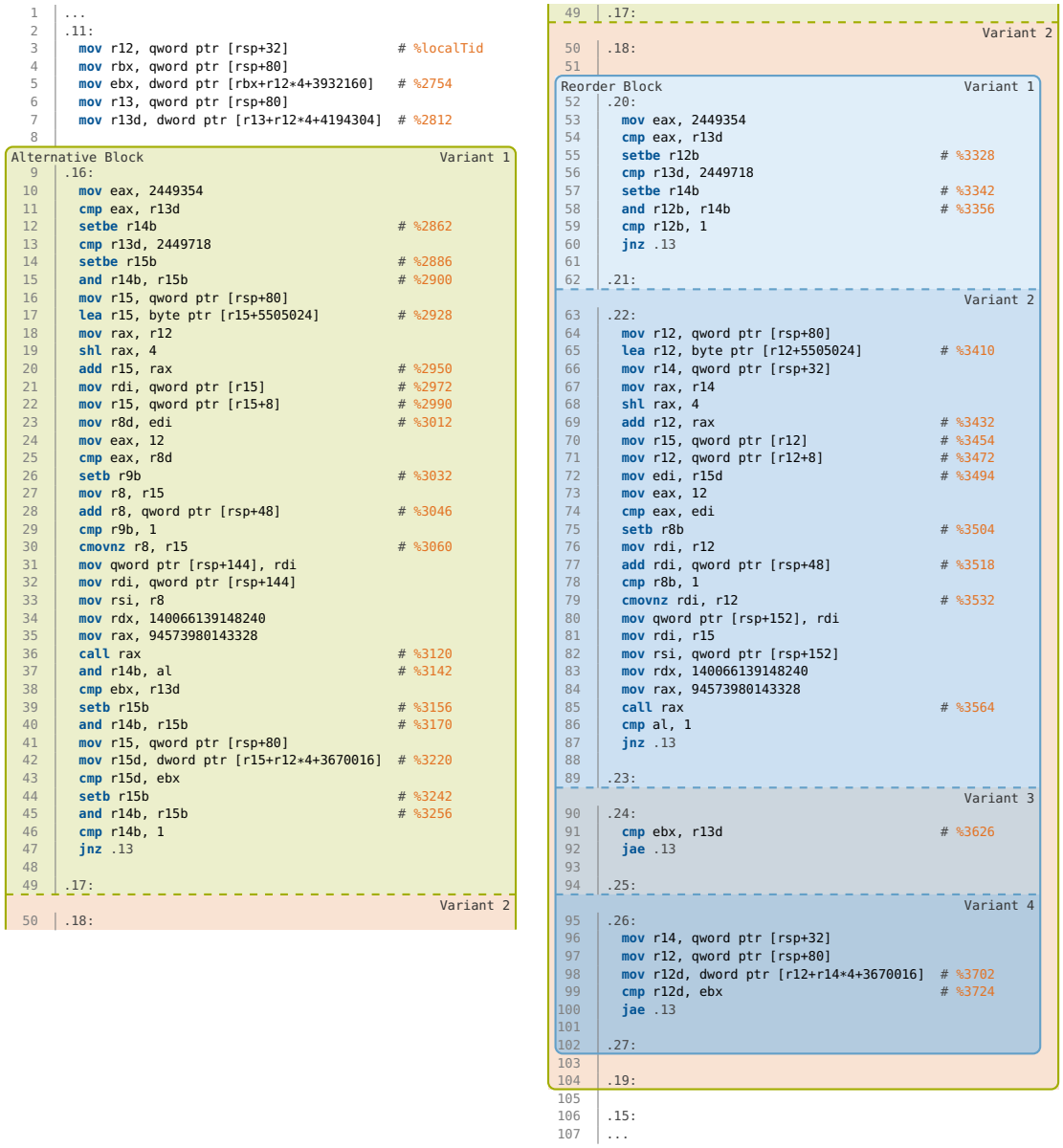


Figure 5.1.: Machine code with dynamic blocks for the Umbra IR from Figure 4.7 (TPC-H Q12) generated by the variant-aware version of the Flying Start compiler. The comments after the assembly instructions denote the IR value computed by the instruction(s).

Hence, our version of the Flying Start backend compiles functions with dynamic blocks so that the dynamic blocks' semantics still apply to the assembly code, i.e., it is possible to reorder and exchange fragments of the generated instructions. Of course, the difficulty now is to emit machine code that computes the expected result and can be modified later. To do so, we have to solve the following three problems:

1. Umbra's block placement algorithm optimizes the block order for tight loops. In particular, it could move basic blocks from a dynamic block to any position in the program. For assembling the variations later in Section 5.2, we have to avoid this situation and keep all basic blocks in the variant they belong to.

2. The algorithm for computing the lifetime of IR values (liveness analysis) does not take dynamic blocks into account. This results in an underestimation of the lifetime of values in reorder blocks and incorrect register assignments.

3. The register allocation also has to consider dynamic blocks: When switching between the variants of a dynamic block, we have to restore the register assignment.

5.1.1. Block Placement

The Flying Start compiler implements its own block placement algorithm to generate tight loops and facilitate range-based liveness analysis. The algorithm is based on a simple principle: first, identify all loops in the program and then place the blocks that belong to the same loop one after the other. The challenging part of the algorithm is to find the loops in linear time. Umbra solves this problem by computing the loop nesting forest using Tarjan's algorithm [54]. One should note that the algorithm only works for programs with a reducible control flow graph [22], i.e., all loops are entered solely through one basic block, the loop header. The operator implementations ensure that only natural loops are generated, avoiding irreducible graphs during query compilation.

To facilitate the compilation of dynamic blocks and the later assembling of the variations, we restrict the block placement as follows. All basic blocks that belong to the same dynamic block must be placed together. However, this limitation collides with our initial goal of generating tight loops. For instance, consider the control flow graph shown in Figure 5.2. The basic blocks 2, 3, 4, 6, and 7 are all part of a loop. Apart from the loop header and tail, blocks 4 and 6 are also part of a dynamic block. The dynamic block has an additional third basic block that is not in the loop: block 5. The block placement algorithm would place the basic blocks of the loop consecutively, and block 5 is moved after the loop's tail (block 7) and outside of the dynamic block. Hence, the dynamic block would be split into two parts.

This problem can be solved by modifying the algorithm for identifying the loops. We introduce a virtual loop for every dynamic block that contains all basic blocks in the dynamic block and thereby enforce the block placement in dynamic blocks. For the

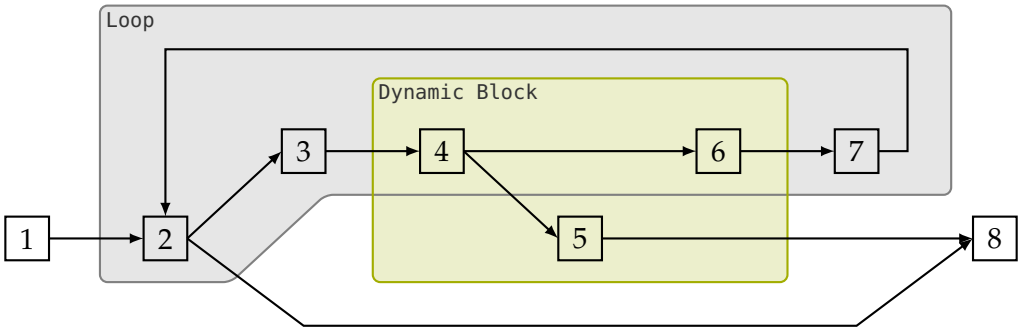


Figure 5.2.: Dynamic Block inside of a loop.

given example, the algorithm first generates a virtual loop consisting of blocks 4 to 6 and then adds this loop with all its basic blocks to the actual loop. The existing placement algorithm rearranges the basic blocks without splitting the dynamic blocks and places the blocks 4, 5, and 6 next to each other.

However, this implementation introduces a new limitation: we cannot interleave dynamic blocks and loops, i.e., loop entry and tail (last block in the loop) must be in the same dynamic block. Hence, a dynamic block completely encloses a loop, or it is fully enclosed by a loop, but situations where the loop head is part of a dynamic block and the tail not (or vice versa) must be avoided. As for the reducible control flow, we manually ensure that our operator implementations do not interleave dynamic blocks and loops.

5.1.2. Liveness Analysis

The Flying Start compiler computes the lifetimes of IR values in linear time. As proposed by Kohn et al., the liveness is not computed per basic block, but instead, the compiler uses an interval-based approach to avoid quadratic runtime [33]. For every IR value, it stores the interval where the value must be alive. Usually, the end of the interval is the last basic block that reads the value, but in some cases, we have to extend the lifetime beyond this block. Again, consider the control flow graph in Figure 5.2: if a value is defined in block 1 and last read in block 3, the value's life range must be extended to the last block of the loop (block 7). If the interval ends in block 3, a subsequent block in the loop can override the value's register, and the value cannot be accessed in the next iteration of the loop. Therefore, we extend the lifetime of an IR value to the end of the enclosing loop of the last basic block that reads it.

A similar situation can occur for reorder blocks: take, for example, an IR value defined before the dynamic block and last read in the first variant. As for loops, the live range of the value must be extended to the end of the reorder block since the first variant can be executed after the other variants. Similarly, we also have to consider the situation

where the value is defined inside a reorder block and last accessed outside the dynamic block. Here we extend the beginning of the value's lifetime to the start of the reorder block. For alternative and optional blocks, the live ranges are not modified as the order in which the variants are executed remains the same.

5.1.3. Register Assignment

Lastly, we modified the register assignment logic used in the final pass over the intermediate representation while emitting the assembly instructions. At the beginning of the function, we reserve space on the stack for all IR values. Values with disjoint lifetimes can share stack entries to reduce memory usage and keep them in the L1 cache. Nonetheless, to achieve good performance, it is not sufficient to keep the values in the L1 cache; we must also use the CPU's registers. Umbra's register assignment algorithm determines for every IR instruction in which CPU registers the operands are loaded and where to place the result.

The algorithm partitions the set of CPU registers into three classes. The scratch registers hold temporary values that are only used by the current IR instruction. In fixed registers, we store IR values needed in multiple blocks, which cannot be evicted until the last instruction reads the value. The remaining unfixed registers cache IR values used by more than one instruction. Values in the unfixed CPU registers are regularly evicted and must be reloaded from the stack.

The Flying Start compiler emits assembly instruction in a single pass. It starts by compiling the first basic block of the function and then continues with the next block in the order determined by the block placement algorithm (cf. Subsection 5.1.1). While compiling the basic blocks, we track the IR values which are loaded into the registers. When switching from the current basic block to the subsequent one, most of the time, we can keep the register assignment intact and reuse IR values loaded before[1]. However, for dynamic blocks, we have to modify the register assignment when entering/leaving a dynamic block or switching between the block's variants.

For instance, consider the machine code for TPC-H Q12 in Figure 5.1 at line 49 when switching from the first variant of the alternative block to the second variant. At this position, the Flying Start compiler has filled the CPU registers with IR values needed for evaluating the predicates. But, if we now switch to the next variant in the dynamic block, the cached values cannot be reused since only one code fragment will be executed. Hence, we must revert the changes made to the register assignment by the first variant and reset to the state before entering the dynamic block. Similar problems occur for the optional and the reorder block. The following actions are required for each dynamic block:

[1]Some registers must be evicted if multiple blocks can enter the subsequent basic block or the entering block is not the current block, i.e., the current block ends with a jump instruction.

Alternative Block: Before entering the dynamic block, we spill all modified registers and save the current register state. When switching between the variants, we evict all newly allocated registers, i.e., all IR values loaded from the stack or computed after entering the alternative block. We also have to clear the unfixed registers when leaving the dynamic block since the variants could change them. Hence, only the fixed registers keep their IR values after the alternative block, and the other CPU registers are empty.

Optional Block: We only have to modify the register assignment for this dynamic block when leaving the first and only variant. As before, we evict the newly allocated registers, but this time, we restore the original register state with all unfixed registers. Compared to the eviction policy for the alternative block, this approach avoids register movements when the optional block is not used and prefers the execution path without the dynamic block.

Reorder Block: As there is no fixed order in which the variants are executed, we clear all unfixed registers before entering the dynamic block. Similarly, when switching the variant or leaving the dynamic block, we reset the register assignment state so that only fixed IR values from before the reorder block are available. Hence, at the start of a variant and after the reorder block, only the fixed IR values are cached in the CPU registers, and other values must be loaded from the stack.

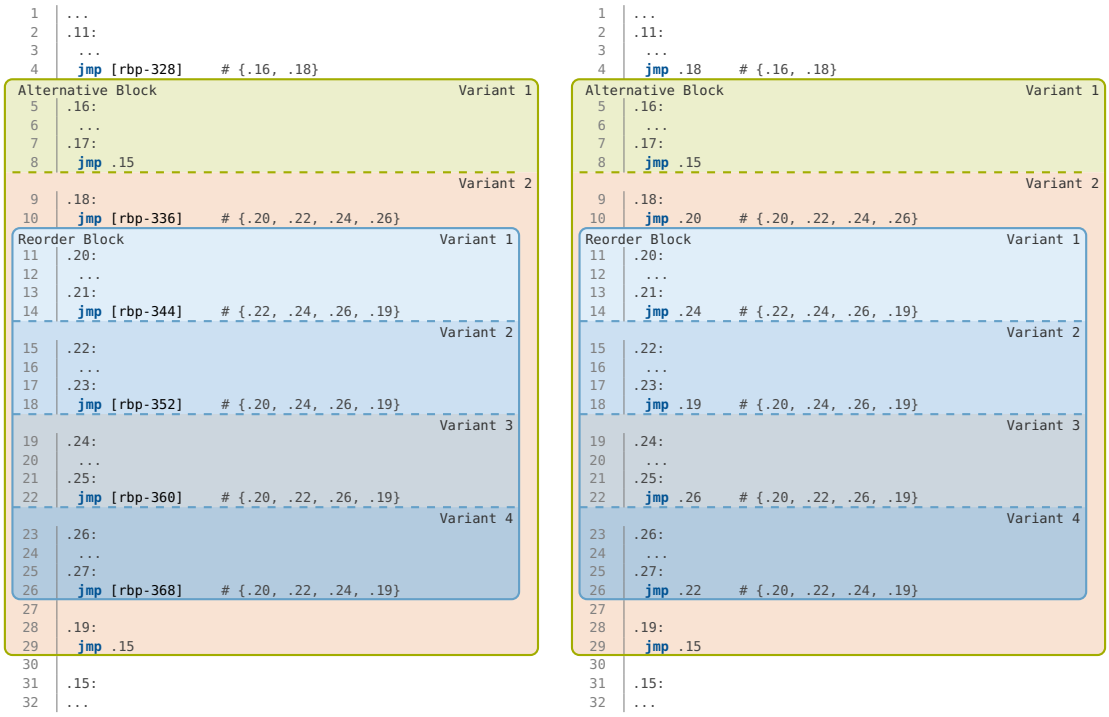
Despite this more limited register assignment, we still generate highly optimized assembly code. In our example in Figure 5.1, the registers `ebx` and `r13d` hold the IR values for the `l_commitdate` and `l_receiptdate` columns. The two values are loaded into the CPU registers in block 11. We access them multiple times in both dynamic blocks, and fixed registers store the values to avoid unnecessary memory loads.

5.2. Rewriting Compiled Queries

With our modifications to the Flying Start compiler, it is possible to compile functions in Umbra's intermediate representation with dynamic blocks. The generated assembly code can still be reordered and exchanged, allowing us to apply the semantics of our Dynamic Blocks framework also to low-level machine code. The last step is implementing a mechanism that performs these dynamic changes on generated code fragments and rewrites the function. Since we might have to assemble several dozens of different variants, we need a mechanism that is both fast and scales to multiple threads.

5.2.1. Jump-based Approach

Our first idea for implementing this rewriting mechanism was to introduce jump instructions before entering a dynamic block and between the variants. If we want



(a) memory-indirect jumps (b) instruction pointer-relative jumps

Figure 5.3.: Jump-based rewriting for TPC-H Q12. The comments after the jump instructions list the possible destinations.

to execute a different variant of the code, we simply change the destination of the jumps and whereby modify the control flow. Figure 5.3a sketches this approach: we use memory-indirect jumps that load the destination address from the stack to switch or reorder variants. The alternative block needs one jump before entering the dynamic block that can either continue execution in the first or second variant. At the end of the first variant, we placed another jump instruction to skip the second variant. For the reorder block, we need more indirect jumps: at the entry of the dynamic block and after every variant. In total, the function needs six memory-indirect jumps to represent all 25 variations. The destination address for each jump is computed upfront and placed in an array. A pointer to this array is passed to the function as an additional argument, and in the function's preamble, we copy this array onto the stack to access the jump addresses efficiently.

If we want the generate the variation with optimal predicate ordering from Figure 4.3b, we place the memory addresses of the following blocks in the given order in the array: `[.18, .20, .24, .19, .26, .22]`. The jump before the alternative block skips the first

variant and continues in the second variant at block 18. After that, execution goes on in block 20 as usual, but at the end of the variant in block 21, we skip the second variant of the reorder block and jump to the third variant starting at block 24. Variant 3 jumps to variant 4, and from block 27, we jump back to the second variant to block 22. Lastly, block 23 leaves the reorder block and goes to block 19.

This approach scales well to multiple execution threads and, apart from generating the vector of the memory addresses, has no overhead for generating different variations of the function. By copying the addresses onto the stack, we can use the base pointer `rbp` for addressing them and do not reserve an additional register for the pointer to the array. Furthermore, we align the first instruction in the jump targets to 16 bytes to facilitate instruction decoding [17]. Nonetheless, memory-indirect jumps can stall the CPU's execution pipeline and affect the performance.

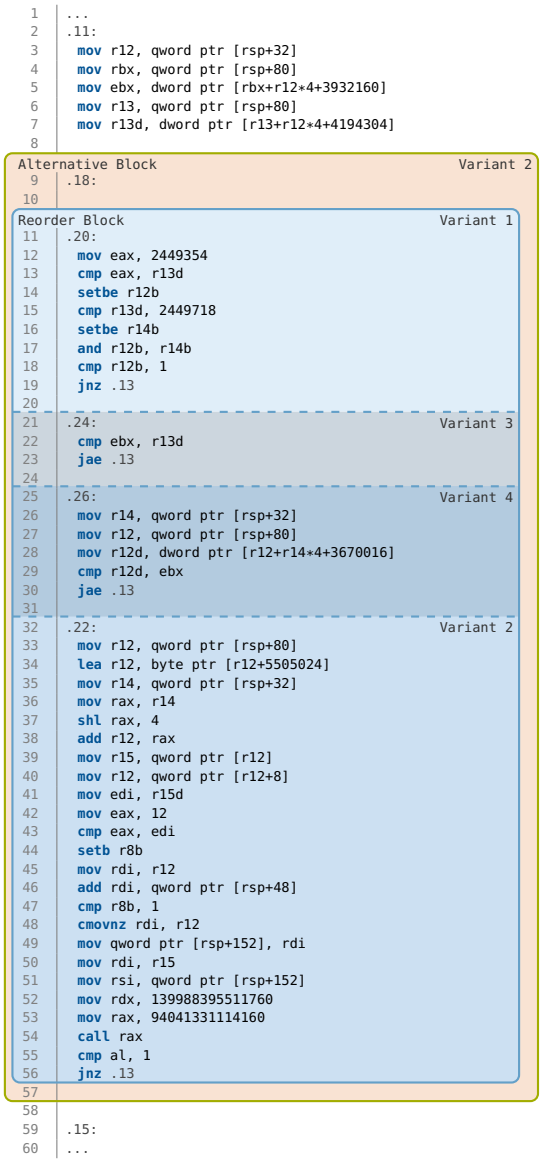
We, therefore, propose a second implementation using direct jumps with offsets relative to the instruction pointer. Instead of loading the destination addresses from the stack, we patch the offsets in the jump instruction before executing the function. Figure 5.3b illustrates this approach. As for the memory-indirect approach, we changed the execution order to the optimal variations. But this time, no memory loads are required, and the destination is encoded into the instruction. With this approach, we first compute the destination addresses for the current variation as before and then replace the old offsets in the executable code with the new ones. However, when executing different variations on multiple threads in parallel, each thread needs its own copy of the function that can be modified.

5.2.2. Direct Approach

While the jump-based approach is easy to implement, causes only a small overhead upfront, and scales well to multiple threads, the additional jumps can cause a significant overhead while executing the function. For instance, adding a jump instruction to the third variant in the reorder block of TPC-H Q12 increases the number of instructions for the variant by 50 %. Furthermore, the jumps interrupt the instruction decoding and put more pressure on the CPU's front-end.

Hence, we devised a direct approach for rewriting the function with dynamic blocks that does not introduce additional jumps. Since we know which code fragments are executed and their order before calling the function, we can generate optimized code for each variation. To do this, we first copy the code fragments into a new memory area using `memcpy`. Fragments that are not executed are skipped, and the reorder blocks' variants are already placed in the correct order. After assembling the specialized version of the function, we perform a second pass to patch instruction pointer-relative offsets in jump and call instructions.

We examine the quality of the generated machine code for the optimal variation of TPC-H Q12 in Figure 5.4. The left side shows the code produced by our modified



(a) with dynamic blocks

```
...
.11:
  mov r12, qword ptr [rsp+32]
  mov rbx, qword ptr [rsp+80]
  mov ebx, dword ptr [rbx+r12*4+3932160]
  mov r13, qword ptr [rsp+80]
  mov r13d, dword ptr [r13+r12*4+4194304]

.20:
  mov eax, 2449354
  cmp eax, r13d
  setbe r14b
  cmp r13d, 2449718
  setbe r15b
  and r14b, r15b
  cmp r14b, 1
  jnz .13

.24:
  cmp ebx, r13d
  jae .13

.26:
  mov r13, qword ptr [rsp+80]
  mov r13d, dword ptr [r13+r12*4+3670016]
  cmp r13d, ebx
  jae .13

.22:
  mov rbx, qword ptr [rsp+80]
  lea rbx, byte ptr [rbx+5505024]
  mov rax, r12
  shl rax, 4
  add rbx, rax
  mov r13, qword ptr [rbx]
  mov rbx, qword ptr [rbx+8]
  mov r14d, r13d
  mov eax, 12
  cmp eax, r14d
  setb r15b
  mov r14, rbx
  add r14, qword ptr [rsp+48]
  cmp r15b, 1
  cmovnz r14, rbx
  mov rdi, r13
  mov rsi, r14
  mov rdx, 139497207448144
  mov rax, 93929822198432
  call rax
  cmp al, 1
  jnz .13

.15:
...
```

(b) without dynamic blocks

Figure 5.4.: Assembly code for the optimal variation for TPC-H Q12 (cf. Figure 4.3b) generated by the variant-aware Flying Start compiler for dynamic blocks with direct rewriting (left) and the original version (right).

version of the Flying Start compiler for dynamic blocks in combination with the direct rewriting approach. The assembly code for the variation compiled with the original version of the Flying Start backend is displayed on the right side. The code generated by the two versions is almost identical, aside from the register assignment. Furthermore, the version with dynamic blocks performs two additional memory loads to retrieve the local tuple id from the stack. Overall, our modifications to the Flying Start compiler and, in particular, to the register assignment degrade the quality of the generated machine code slightly.

Admittedly, the example we considered in this chapter is simple and does not use ϕ nodes to extract values from the dynamic blocks. For more complex functions, we also noticed only a minor overhead when using direct rewriting. However, since assembling the query is now more expensive than for the jump-based approach, it is too expensive to rewrite the function every time before calling it. Hence, we cache the assembled code for each variation and reuse it for later executions. The assembly code for a variation is generated by the first thread that tries to execute it. Other threads that want to run the same variations have to wait until the first thread finishes assembling it. With multiple threads, it is possible to assemble different variations in parallel and insert them into the cache.

5.2.3. Comparison

We close this chapter with an experiment that evaluates the overhead of the jump-based mechanism compared to the direct rewriting approach. We use the TPC-H, TPC-DS, and SSB datasets at scale factor 10 and the join order benchmark with their respective queries. The benchmark reports the per pipeline speedup/slowdown using (indirect) jumps relative to the direct rewriting approach. The speedup is computed as follows: First, we determine the average runtime for a pipeline per variation. Then, the ratio between the two rewriting approaches is computed for every variation, and, finally, the average over the per variation speedups in the same pipelines is returned as the per pipeline speedup.

Figure 5.5 shows the distribution and the mean value of the per pipeline speedups using memory-indirect and instruction pointer-relative jumps. The first plots determine the overhead for each of the three dynamic optimizations individually, and the last two plots show the speedup when combining all optimizations. As expected, the jump-free version of the pipeline assembled with the direct rewriting approach is faster than the jump-based approach. The pipelines with memory-indirect and instruction pointer-relative jumps are on average 5 % slower.

Surprisingly, the instruction pointer-relative jumps are slower than the memory-indirect jumps. We assume that the branch prediction is responsible for this behavior: The CPU's branch target buffer predicts the destination address of the memory-indirect jumps based on the current position in the code [17]. A jump relative to the instruction

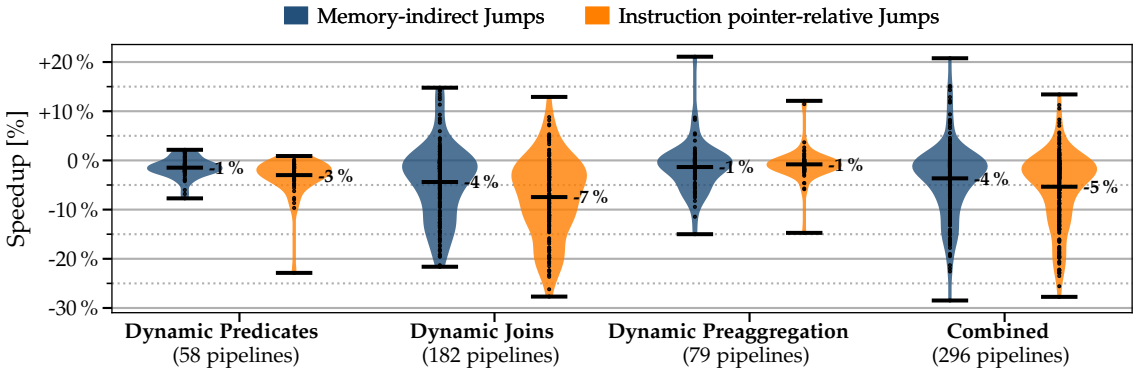


Figure 5.5.: Relative performance of the jump-based approach. The direct rewriting mechanism without any additional jumps serves as baseline.

pointer, on the other hand, must first decode the offset and compute the destination address.

The memory-indirect approach computes the destination addresses for the jumps upfront and places them in an array that is passed to the generated code. For the performance of the code, it is essential to copy this array onto the stack and access the elements using the stack pointer. Another important factor is the alignment of the jump targets to 16-byte. We ran the experiment also with unaligned jump addresses and observed an additional 2 % slowdown.

Although our rewriting approach performs best, memory-indirect jumps are still interesting. Assembling a specialized version of the machine code is more complex and requires changes to the underlying compiler. The memory-indirect approach can also be implemented on top of LLVM using the `indirectbr` instruction.

6. Dynamic Execution

In the previous chapters, we discussed code generation with our Dynamic Blocks framework and the compilation with the Flying Start backend. The last missing piece for adaptive query processing is the execution of functions with dynamic blocks. Like Vectorwise or NoisePage, we have to balance the exploration and the exploitation phase. During exploitation, different variations of a function are executed, and runtimes are recorded. The exploitation phase then chooses the best-performing variation and executes it.

Both Vectorwise and NoisePage re-evaluate different variations in constant intervals to adapt the executed variant to changing data distributions. However, interleaving the exploration and exploitation phase is not compatible with Umbra's existing execution model. Adaptive Execution first runs a slow version of the function generated by the Flying Start compiler and later switches to an optimized version of the function that LLVM compiles in the meanwhile [34]. This technique allows Umbra to hide LLVM's slow compilation times and keep the otherwise idle threads busy during compilation. However, exploring different variations is only possible while executing the machine code generated by the Flying Start compiler. Once LLVM finishes compiling, a static version of the function without dynamic blocks will be executed. Hence, we plan to use our Flying Start compiler with dynamic block support for finding the best variation and then compile it with LLVM to generate optimized machine code. Combining dynamic blocks and LLVM is out of scope for this thesis and is considered future work. Instead, we focus on finding the best variation with as few exploration runs as possible.

6.1. Switching Between Variations

Umbra uses morsel-driven execution in combination with work-stealing to parallelize query processing. As proposed by Leis et al., we divide the input to a pipeline into small chunks of work, so-called morsels [36]. The pipeline's function obtains a morsel and processes all tuples in the morsel. Once a thread finishes processing a morsel, it asks the scheduler for the next morsel and executes the function again. This approach allows for a simple parallelization scheme where each thread evaluates the pipeline until no tuples/morsels are left. For table scans, a single morsel contains between 1000 and 20000 tuples.

Since morsels are the smallest unit of work in Umbra's query engine, we can change

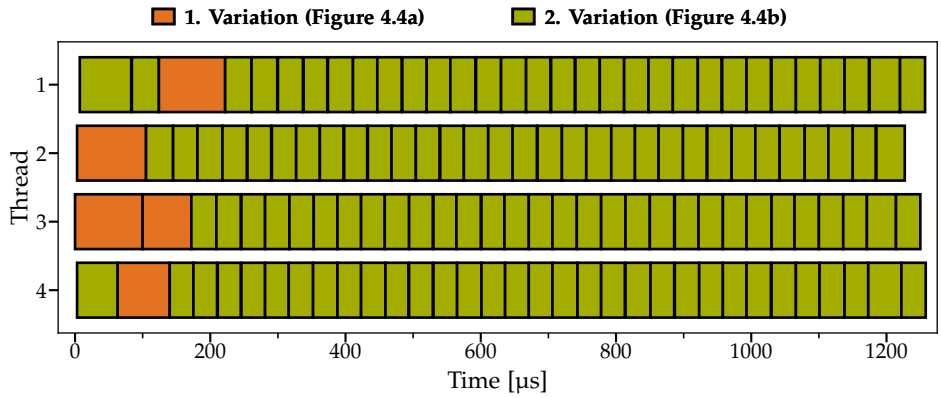


Figure 6.1.: Execution Trace of TPC-H Q2 (scale factor 1) using 4 threads for the third pipeline.

the executed variation only when switching the morsel. Before processing a morsel for a pipeline with dynamic blocks, we first determine which variation to execute. We alternate the variations during the exploration phase until the best-performing variation can be determined. The exploitation phase executes the optimal variation found before. Once a variation is chosen, we retrieve the machine code from the Flying Start compiler and call it with the current morsel. The Flying Start compiler assembles the code using the direct rewriting approach (cf. Subsection 5.2.2).

Figure 6.1 shows an execution trace of the morsels in pipeline 3 of TPC-H Q2. For this query, it is possible to reorder the two hash joins, resulting in the two variations from Figure 4.4. We evaluate both variations five times each and choose the one with the minimum runtime for a single morsel. In the case of Q2, reordering the joins as in Figure 4.4b provides better performance, and we switch to the reordered version for exploitation. Note that the exploitation phase starts once the first ten morsels are issued. Although the exploration phase is not finished at this point, we use the variation with the lowest measured runtime till then and switch later to a faster variation.

6.2. Choosing the Best-performing Variation

In the previous section, we used the minimum runtime for a single morsel to find the best-performing variation. However, this metric is susceptible to data skew. Therefore, we also considered other policies for identifying the best variation. Both the average and median runtime per variation are easy to implement and take all measurements into account.

For all three metrics, we implement data structures that record the runtimes and compute the best-performing variation. Furthermore, the structure also determines

which variation to execute next and when to start the exploitation phase. We use the `rdtsc` instruction to retrieve a hardware timestamp and report the difference between the timestamps before and after executing the pipeline as the morsel's execution time. The length of the exploration phase is determined by two factors: 1) the number of variations n and 2) the number of measurements per variations p. Hence, we evaluate the different variations for the first $n \cdot p$ morsels and then exploit the best-performing one.

For TPC-H Q2, we evaluated two variations with $p = 5$ measurements and explored them in only ten morsels. However, a query with more variations, like TPC-H Q12, will spend far more time in the exploration phase. Furthermore, evaluating a slow variation can have a significant overhead. Therefore, we split the exploration phase into runs and constantly reduce the number of different variations to explore. We eliminate variations with high execution times early on and evaluate the most-promising ones more often. The best-performing variation is determined based on the minimum, average, or median runtime.

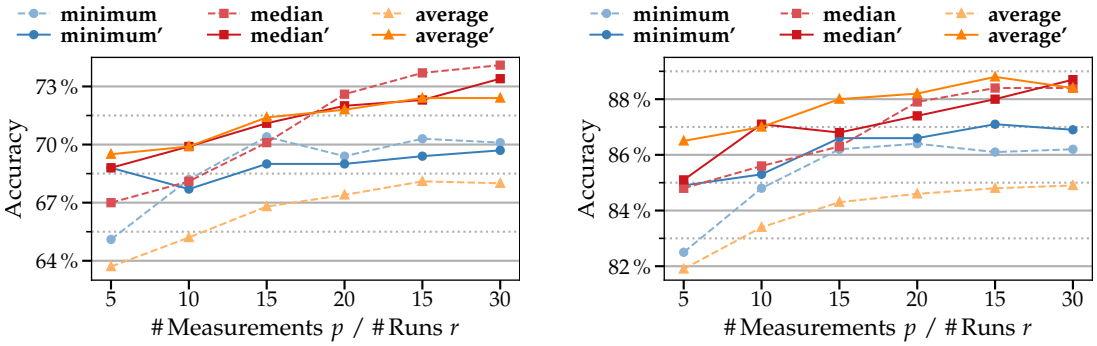
The number of variations we consider in each run decreases exponentially until only one is variation is left. We evaluate at least as many morsels as there are threads per run to fully parallelize the exploration phase. If all morsels in the current run were scheduled, but some threads are still running, we preemptively explore variations for the next run on the idle threads. The number r denotes the number of runs to perform.

6.3. Comparison

Umbra with Dynamic Execution implements six different policies for choosing the best-performing variation. The first three policies measure each variation p-times and choose the variation with the best minimum, average, median execution time. We further refined these policies by eliminating slow variations early on using the run-based exploration phase from Section 6.2. We refer to the optimized version as **minimum'**, **average'**, and **median'**.

We evaluate the effectiveness and accuracy of the six policies as follows: First, we execute each variation for a pipeline individually and determine the optimal implementation of the query. Then, we evaluate the pipeline again, but this time the Dynamic Execution framework explores different variations during execution and chooses a variation using one of the six policies. Based on these measurements, we compute the accuracy of the policies as the percentage of executions that choose the best-performing variation from the first pass. Figure 6.2a reports the results for different values of the parameter p respectively r that determine the number of measurements per variation and the number of runs.

All six policies perform similarly and choose the best-performing variations for roughly 70 % of the executions. The **median** policy achieves the highest accuracy for



(a) optimal variation

(b) faster than the default implementation

Figure 6.2.: Accuracy of the different policies for choosing the best-performing variation or one that is at least as fast as Umbra's default implementation.

30 measurements per morsels. However, in practice, the exploration phase is too long to exploit the found variation effectively. We, therefore, find that the **average'** and the **median'** policy perform best for realistic values for the parameter r (10, 15, or 20).

While finding the optimal variation is crucial, overall, it is more important to avoid performance degradations. Hence, we performed a second experiment investigating how often Dynamic Execution chooses a variation that is at least as fast as Umbra's default implementation. This time the policies choose a good variation in more than 85 % of the cases (cf. Figure 6.2b, for $r >= 10$). Overall, the **median'** policy performs best with more than 88 % accuracy with 30 exploration runs. For fewer runs, the **average'** policy performs better: with 15 runs, an accuracy of 88 % is achieved. Although the accuracy decreases by 1 %, we decided to use the **average'** policy with $r = 10$ in the following benchmarks. This setting evaluates slow variations less often and offers a better tradeoff between the length of the exploration phase and accuracy. We also decided against the **median'** policy as it is more expensive to compute the median value than the mean.

7. Evaluation

We now evaluate our Dynamic Blocks framework and the Dynamic Execution strategy in end-to-end benchmarks on the TPC-H, TPC-DS, SSB, and JOB datasets. We first look into the compilation and execution overhead of our Dynamic Blocks framework and then investigate where Adaptive Query Processing improves or reduces performance in the end-to-end benchmarks. For this, we analyze 16 queries in detail and examine the factors that cause speedups or slowdowns.

We run the benchmarks on an Intel i9-7900X CPU (Skylake X, 3.3-4.4 GHz) with 10 cores (20 threads) and 128 GB of memory. The machine has Ubuntu 21.04 installed (Kernel 5.11) and uses gcc 10.3 and llvm 12.0 for compilation. All modifications were made to Umbra at version 9f134bc7. In Appendix A, the recorded runtimes for the five versions of Umbra are listed, and Appendix B shows the query plans for 16 queries we investigate further. Unless stated otherwise, we use the TPC-H, TPC-DS, and SSB dataset at scale factor 10 and repeat all measurements a hundred times.

In the experiments, we compare the unmodified version of Umbra (**Umbra**) against the adaptive version **Umbra**$^{\mathbf{AQP}}$ with the three dynamic optimizations and our modified Flying Start compiler with dynamic blocks. Umbra without Adaptive Query Processing uses the original Flying Start backend that generates the code without the Dynamic Blocks framework. Our variant-aware version of the Flying Start backend employs the direct rewriting approach from Subsection 5.2.2, and Dynamic Execution uses the **average'** policy with 10 runs for switching between variations (cf. Section 6.2). When evaluating one of the optimizations individually, we denote them as follows: **Umbra**$^{\mathbf{Pred}}$ (Dynamic Predicates, Subsection 4.5.1), **Umbra**$^{\mathbf{Join}}$ (Dynamic Joins, Subsection 4.5.2), and **Umbra**$^{\mathbf{PreAgg}}$ (Dynamic Preaggregation, Subsection 4.5.3).

7.1. Theoretic Performance Boost

Before starting the experimental analysis, we first determine the theoretical performance gain of our three dynamic optimizations without considering the execution and compilation overhead caused by the Dynamic Blocks framework. For this, we compare the runtime of Umbra's default implementation against the optimal variation compiled with the original version of the Flying Start backend without dynamic blocks support and report the speedup per query. Since both the optimal and the default variation are compiled with the same backend and do not use Dynamic Execution, we can filter out

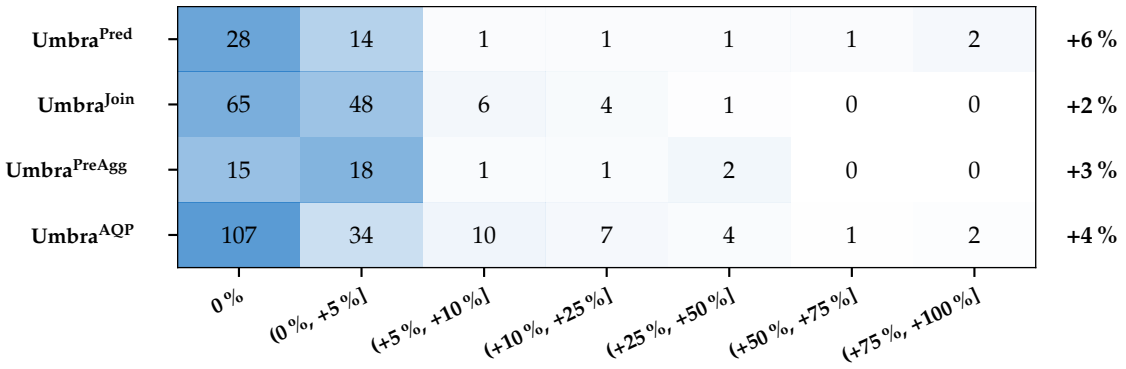


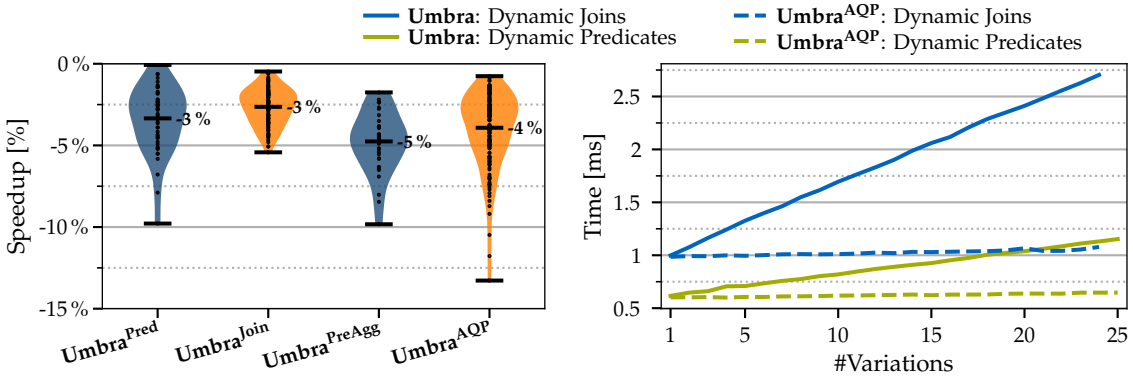
Figure 7.1.: Histogram of the possible relative performance boost of **Umbra**$^{\mathbf{AQP}}$ over **Umbra** without considering the execution and compilation overhead caused by the Dynamic Blocks framework.

the overhead of the Dynamic Blocks framework.

Figure 7.1 shows the distribution and the average of the possible speedups. Umbra already chooses the optimal variation for roughly 60 % of the queries, and the performance cannot be improved further using one of the three optimizations. For the remaining queries, a different variation is used, but we observe only for 15 % of the queries a significant performance improvement (>5 %): adapting the predicate evaluation, in particular, can reduce the execution time by factor 2. On average, the three optimizations increase the performance by 6 %, 2 %, and 3 %, resulting in a total speedup of 4 %. Of course, this experiment only determines an upper bound for the possible speedup as finding the optimal variations at run-time causes a non-neglectable overhead. Nevertheless, the results are a good baseline to evaluate the effectiveness of Adaptive Query Processing in Umbra.

7.2. Compilation Overhead

Our Dynamic Blocks framework and the variant-aware version of the Flying Start compiler come at a prize: during code generation, the different variants must be constructed, and lowering Umbra IR to assembly instructions is more complex. On the other hand, our dynamic rewriting approach makes it possible to efficiently assemble the variations with almost no execution overhead. Figure 7.2 investigates the impact of our modification to Umbra's code generation phase closer: The plot on the left side visualizes the performance of our variant-aware Flying Start compiler relative to the original version without dynamic blocks. The second plot shows the time the two backends need to assemble the variations for a pipeline with Dynamic Predicates



(a) Relative compile time of our variant-aware Flying Start compiler. Baseline is the original version of the backend without dynamic blocks.

(b) Compilation times with the original Flying Start compiler (solid lines) and our variant-aware version (dashed lines).

Figure 7.2.: Compilation overhead of the variant-aware Flying Start compiler and the original version without dynamic blocks.

(TPC-H Q12) and one with Dynamic Joins (SSB Q42).

As expected, our version of the Flying Start backend is slower for all three dynamic optimizations and their combination. We measure an average slowdown of 4 % and maximum compile time overhead of 13 % when combining multiple optimizations in the same pipeline. Nevertheless, this overhead is acceptable when considering that compiling a query usually takes only a few milliseconds, and most of the time is spent executing the generated code.

The benefits of our approach are then again visible in Figure 7.2b. We extracted the compilation times for the first n variations when using the original Flying Start compiler without dynamic blocks (**Umbra**) and our variant-aware version (**Umbra**$^{\mathbf{AQP}}$). The green lines show our running example from TPC-H Q12 that uses the Dynamic Predicates optimization with 25 variations. For the Dynamic Joins (blue lines), we use Q42 from the star schema benchmark. The query has a chain of four hash joins that can be reordered, resulting in 24 different variations.

The compilation times grow linearly with the number of variations. Assembling the 24 variations for the Dynamic Join with the original compiler takes roughly 1.7 ms. Our version, in contrast, needs less than 0.1 ms to generate the variations. The same happens for the query with Dynamic Predicates: the compilation time with the variant-aware backend reduces by one order of magnitude. Please note that in this experiment, **Umbra** still uses the Dynamic Blocks framework to generate the different variants in Umbra IR; we only changed the underlying backend responsible for emitting the machine code.

In conclusion, our Dynamic Blocks framework and the variant-aware Flying Start backend introduce only a small overhead when compiling queries. The modified

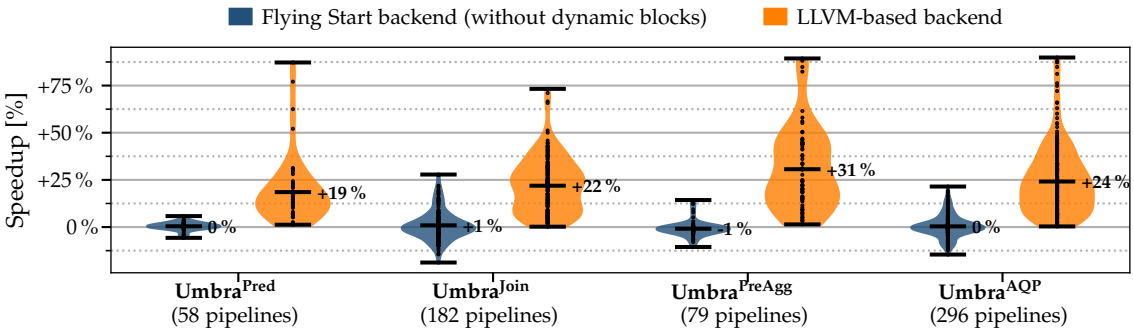


Figure 7.3.: Execution time of the original Flying Start backend and the LLVM-based backend relative to our modified version of the Flying Start backend with dynamic blocks support.

backend generates the variations one order of magnitude faster than the original Flying Start compiler.

7.3. Execution Overhead

After looking into the compilation overhead, we now measure the overhead caused by our modified version of the Flying Start compiler with dynamic block support from Chapter 5. We repeat the experiment from Subsection 5.2.3, but this time the performance is reported relative to the original Flying Start compiler without dynamic blocks and the LLVM-based backend. As before, Figure 7.3 shows the per-pipeline speedups of the two backends. While the LLVM-based backend always generates faster code, there is no clear winner for the original Flying Start backend. We sometimes also observe a slowdown compared to our variant-aware version with dynamic blocks.

Compared to the original version, our modified Flying Start backend spills register values earlier to the stack to keep the register assignments between variants intact. Hence, we sometimes lose performance due to additional loads but in other cases spilling values to the stack and loading a different one is beneficial. As a result, these two factors neutralize each other, and overall, our modifications introduce no significant speedup or slowdown. Both versions of the Flying Start backend produce equally fast code, and we observe no execution overhead for machine code with dynamic blocks.

The experiment also shows that the performance can be increased further by roughly 25 % using the LLVM-based backend. However, this comes at the prize of higher compilation times. Therefore, functions should only be compiled with the LLVM-based backend once the best-performing variation is found as the existing Adaptive Execution framework does. As this feature is out of scope for this thesis, it is considered future work.

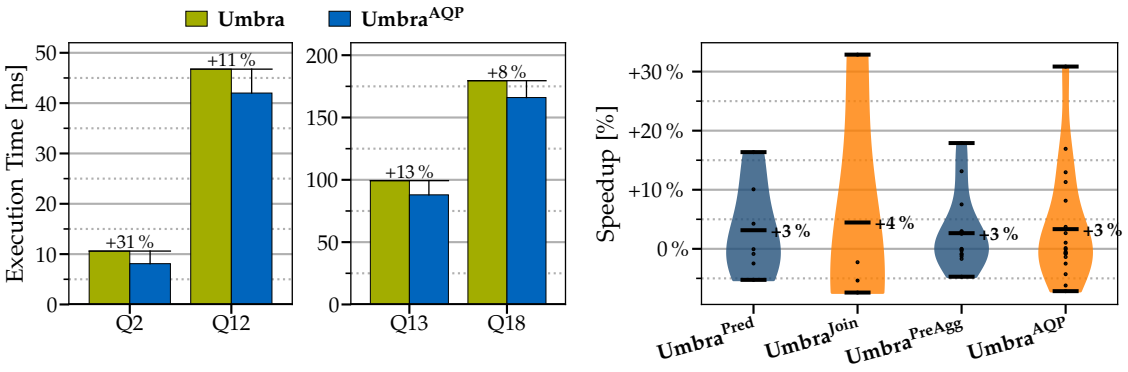


Figure 7.4.: Adaptive Query Processing on TPC-H: execution times and speedup relative to non-adaptive Umbra.

7.4. End-To-End Benchmarks

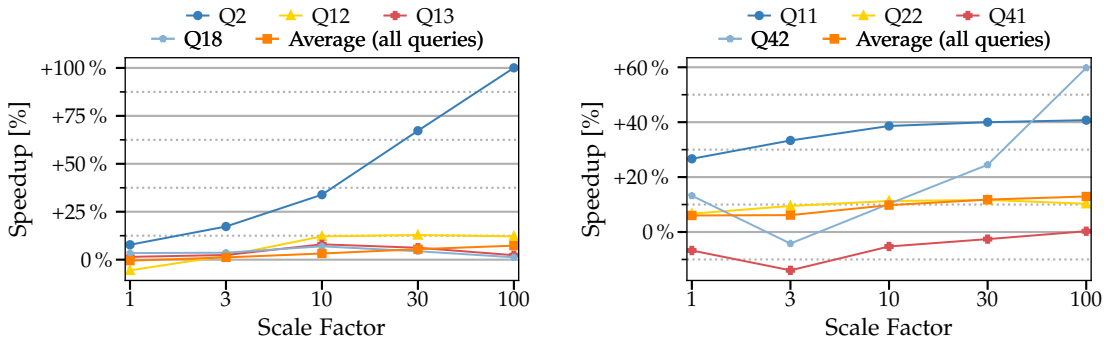
The previous two experiments analyzed the compilation and execution performance individually; our end-to-end benchmarks report the total query runtimes and determine the speedup achieved by Adaptive Query Processing. Furthermore, for each of the four datasets, we inspect four queries closer that show a significant speedup or slowdown and analyze the effects of the three dynamic optimizations. We use the Dynamic Execution framework to find the optimal variation while executing the query.

7.4.1. TPC-H Benchmark

First, we evaluate the TPC-H benchmark. Figure 7.4 shows the overall speedup and the runtimes of the four queries we inspect closer. In total, 19 of the 22 queries use at least one of the dynamic optimizations at scale factor 10. When combining the three optimizations in **Umbra**$^{\text{AQP}}$, we measure an average speedup of 3 %. Overall the dynamic optimizations achieve similar improvements, and four queries show a significant performance boost with more than 10 %.

Q2 We used this query as a running example for reordering hash joins throughout this thesis. It is possible to change the order of the joins with the `eurosupp` and the `part` table (cf. Figure 4.1). Umbra's default implementation, which first joins the `eurosupp` table, executes the pipeline in 6 ms. Our Dynamic Execution framework correctly reorders the joins in all repetitions, reducing the pipeline's execution time to 3.35 ms. Since the pipeline accounts for roughly 60 % of the query's runtime, the optimization improves the overall performance by 1.31x.

In order to analyze the effects of Adaptive Query Execution on smaller and larger



(a) TPC-H (b) SSB

Figure 7.5.: Speedup of **Umbra**$^{\mathbf{AQP}}$ relative to **Umbra** for increasing scale factors.

datasets as well, we repeated the experiments for the scale factors 1, 3, 30, and 100. Figure 7.5a shows the results of these experiments. For Q2, we observe that the speedup increases with the scale factor of the dataset as the query spends more time in the third pipeline. For scale factor 100, the runtime improves by more than 100 %.

Q12 Like Q2, we used the query before as an example for the Dynamic Predicate optimization. With four predicates (cf. Figure 4.2), 25 variations for the table scan on `lineitem` are possible. Umbra with Adaptive Query Processing chooses a variation that changes the evaluation order of the filter predicates. The performance is increased significantly by moving the *in* statement after the two comparisons on the date columns. We observe a performance boost of 5 ms in the adaptive pipeline compared to the default implementation, and the execution time is reduced from 30 ms to 25 ms. In 80 % of the executions, the Dynamic Execution framework chooses the optimal ordering from Figure 4.3b. The remaining executions use a slightly different implementation, where the two comparisons are exchanged. Nevertheless, the runtime improves by more than 4.5 ms. Overall, the **Umbra**$^{\mathbf{AQP}}$ is 11 % faster than **Umbra**.

When increasing the scale factor, the speedup does not increase further. However, the speedup reducesf or smaller datasets, and for scale factor 1, we even see a slowdown. For smaller datasets, Umbra schedules fewer morsels and reduces their size. Therefore, the exploration phase is relatively long, and we cannot hide the Dynamic Execution's overhead.

Q13 & Q18 The two queries use Dynamic Preaggregation for aggregating the `lineitem` table. In both, queries the third access path with the pointer to the last inserted entry (cf. Figure 4.11) reduces the runtime of the pipeline by more than 20 %. The Dynamic Execution chooses in all repetitions the optimal implementation.

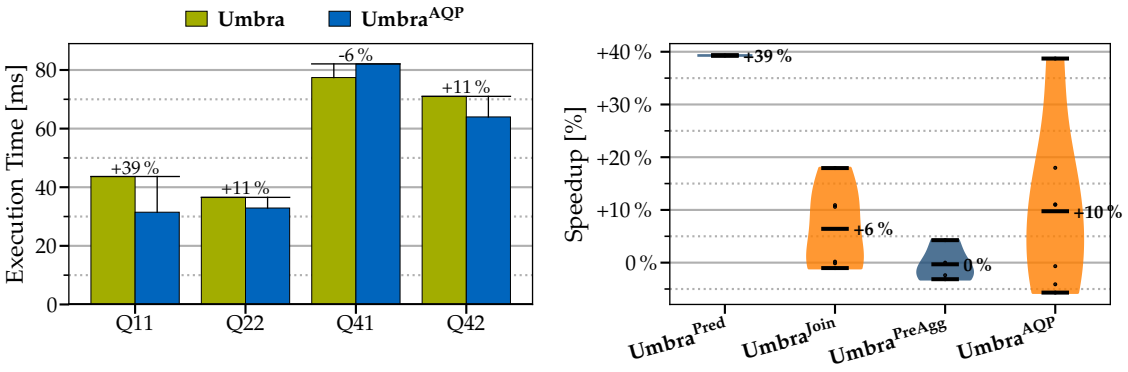


Figure 7.6.: Adaptive Query Processing on SSB: execution times and speedup relative to non-adaptive Umbra.

7.4.2. Star Schema Benchmark (SSB)

For the star schema benchmark, we can use the Dynamic Optimizations in seven of the thirteen queries. With Adaptive Query Processing, the performance of these queries improves on average by 10 % (cf. Figure 7.6). While the highest performance boost is achieved by the Dynamic Predicates that are used in only one query, we still see a 6 % speedup when reordering joins. However, adapting the preaggregation does not help, and we measure a minimal slowdown (less than 1 %) for the optimization.

Figure 7.5b shows how this speedup scales with the size of the dataset. As for the TPC-H benchmark, the benefits of Adaptive Query Processing are evident for large datasets. At scale factors 30 and 100, the performance improves by roughly 12 %, and we observe almost no performance deterioration (<3 %).

Q11 We start with the query with the highest speedup: The performance of Q11 increases by 39 % using by adapting the filter predicates on the `lineorder` table. The table scan evaluates two filter predicates on the 60 million entries of the relation. However, neither of the two conditions is very selective, and branching after one of them causes branch misses. The optimal implementation, therefore, evaluates both predicates together and then branches. As a result, the number of branch misses decreases by factor 3, and the runtime improves by more than 12 ms.

Q22 For this query, we observe an 11 % speedup. However, the performance boost is not caused by one of our dynamic optimizations; instead, our version of the Flying Start accidentally finds a better register assignment and is roughly 4 ms faster than Umbra's default implementation. The query reorders two hash joins, and our backend tends to produce slightly better code than the original Flying Start compiler (cf. Figure 7.3).

Q41 In this query, we lose performance due to the overhead of exploring different variations. In addition, the code generated by the Flying Start compiler for dynamic blocks is slightly slower than compiled with the default version of the compiler.

Q42 The last query we inspect for this benchmark reorders four hash joins. Without Dynamic Joins, the `lineorder` reorder is joined with `customer`, `supplier`, `date`, and `part` (in this order). The Dynamic Execution framework, in contrast, moves the hash join with the `date` table in front of `supplier` and `customer`. Although the reordered join is less selective than the first two, the hash table is smaller, and the number of cache misses decreases. Therefore, the optimal join order is 12 ms faster than Umbra's default implementation. However, finding the optimal variations costs another 5 ms, and the runtime improves only by 7 ms (the worst join order increases the pipeline's execution by more than 3x).

At scale factor 10, reordering the joins improves the runtime by 11 %. For larger datasets, this speedup increases further as the sizes of hash tables for the `supplier` and the `customer` increase linearly with the scale factor. The number of entries in the `date` is independent of the scale factor and the hash table, therefore, remains the same on all scale factors. For the largest scale factor, the performance of Q42 improves by 60 %, and the speedup exceeds the performance benefits measured for the first Q11 (cf. Figure 7.5b).

7.4.3. TPC-DS Benchmark

This experiment evaluates the performance of Dynamic Execution on the TPC-DS benchmark. Figure 7.7 shows the overall speedup and the runtimes for the four queries we inspect closer. In comparison to the star schema benchmark, we now observe a slowdown for the Dynamic Predicates and a speedup of 5 % when optimizing the preaggregation. In total, the performance improves by 2 % on the TPC-DS benchmark at scale factor 10. The maximum performance boost is achieved by Q4 and Q11 using Dynamic Preaggregation.

Q4 & Q11 & Q14a All three queries use Dynamic Preaggregation to adapt the group by operators before the set operations (cf. Table B.3). While Q4 and Q11 achieve a 26 %/38 % speedup, we lose performance in Q14a. The first two queries use eight columns, including string values, as key for the aggregation. Computing the hash value with Umbra's default implementation is therefore expensive and bypassing the lookup table is beneficial. Thanks to this optimization, we save 150 ms and 115 ms in the two queries, and the runtime drops to 600 ms/300 ms. Q14a, on the other hand, uses only three columns as keys in the aggregation and the radix-based hash function reduces the number of hits in the lookup table. Hence, the other access paths allocate more entries

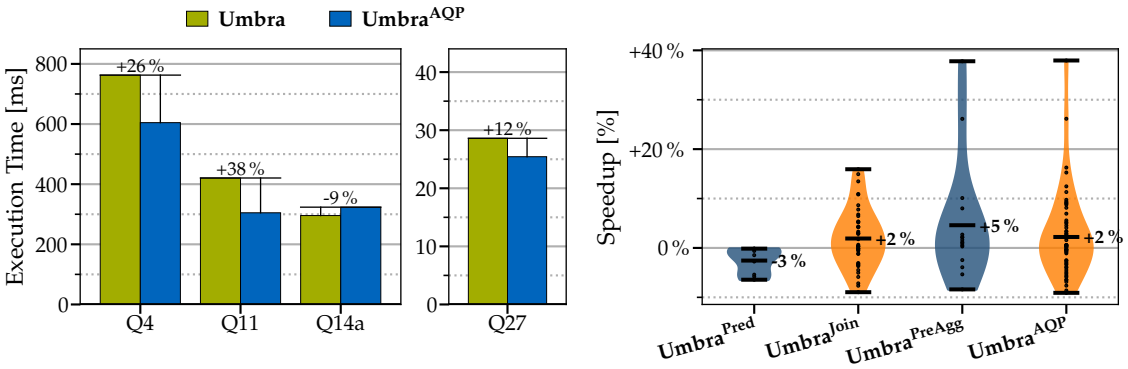


Figure 7.7.: Adaptive Query Processing on TPC-DS: execution times and speedup relative to non-adaptive Umbra.

in the output streams, and the runtime increases by 28 ms as **Umbra**$^{\mathbf{AQP}}$ often chooses a suboptimal implementation.

Q27 The last query we inspect for TPC-DS achieves a 12 % speedup using the Dynamic Joins optimization. We can reorder the three joins in the pipeline starting in `store_sales`. Umbra's query optimizer overestimates the selectivity of the first join with `date_dim` and therefore places it before the second join with `customer_demographics`. In reality, the second join filters 99 % of the incoming tuples while the first join passes one of five tuples on. With Adaptive Query Processing, Umbra finds the optimal join order, and the runtime of the pipeline decreases from 20 ms to 16 ms.

7.4.4. Join Order Benchmark (JOB)

The last end-to-end benchmark looks into the join order benchmark by Leis et al. [38]. Although we use the Dynamic Join optimization in most queries, the highest performance boost is achieved by adapting the filter predicates, as shown in Figure 7.8. With the Dynamic Predicate optimization, the runtime of three queries improves by roughly 70 %. When adapting the join order, we measure for the majority of the queries a speedup/slowdown between -3 % and 3 % caused by the different register assignments of the two versions of the Flying Start backend.

Q13b & Q13d Both queries can reorder the first two hash joins in the pipeline starting in `movie_info` (cf. Table B.4). In this pipeline, Umbra overestimates the selectivity of the first join. However, while in Q13b the estimation error is not large enough the change the join order, in Q13d, a different order is faster. **Umbra**$^{\mathbf{AQP}}$ finds the optimal order for

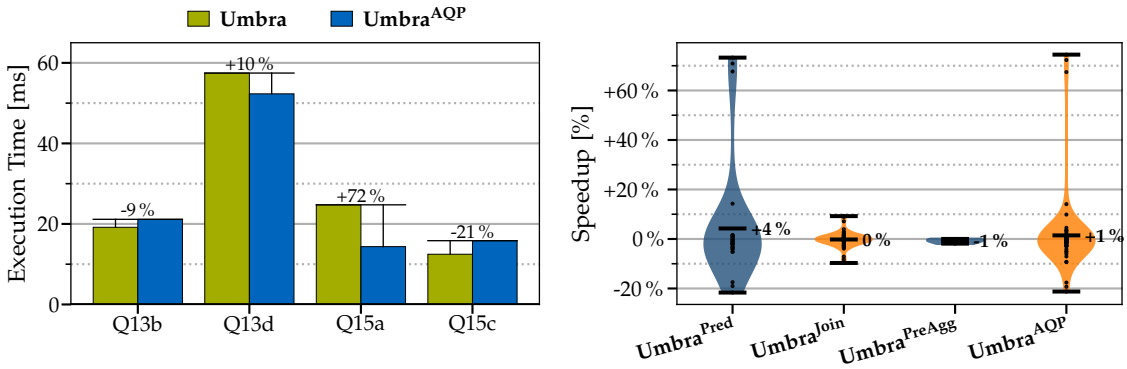


Figure 7.8.: Adaptive Query Processing on JOB: execution times and speedup relative to non-adaptive Umbra.

Q13d, and the execution time improves by 4 ms. But for Q13b, the Dynamic Execution framework chooses the wrong join order resulting in the 2 ms performance loss.

Q15a & Q15c Q15a and Q15c adapt the filter condition on `movie_info` relation. While in Q15a, a different order of the predicates improves the execution time by more than 70 %, Q15c cannot benefit from this optimization. We even lose 3 ms in the pipeline due to the execution overhead introduced by our version of the Flying Start backend. For Q15a, checking for the phrase *internet* eliminates more than 99.9 % of the tuples, and executing it first improves the runtime of the pipeline by 2.4x, resulting in the 70 % speedup. The other two queries with 70 % performance boost (Q15b and Q23b) reorder the same filter condition on `movie_info`.

7.5. Putting It All Together

In the previous section, we saw several queries with a significant speedup and some with an undeniable slowdown. The question now is whether Adaptive Query Processing improves the overall runtime and achieves the theoretical possible performance boost from before (cf. Section 7.1).

Figure 7.9 answers the first part of the question: For all three dynamic optimizations and their combination, we observe on average a speedup in the queries that use the optimizations. Most importantly, the seven queries with more than 25 % speedup from Figure 7.1 provide a similar speedup with Adaptive Query Processing. **Umbra^{Pred}** achieves a 4 % performance improvement and therefore provides the highest speedup of the three optimizations. The reason for this is, to a large extent, the three JOB queries whose runtimes improve by 70 %. Adapting the filter predicates is particularly

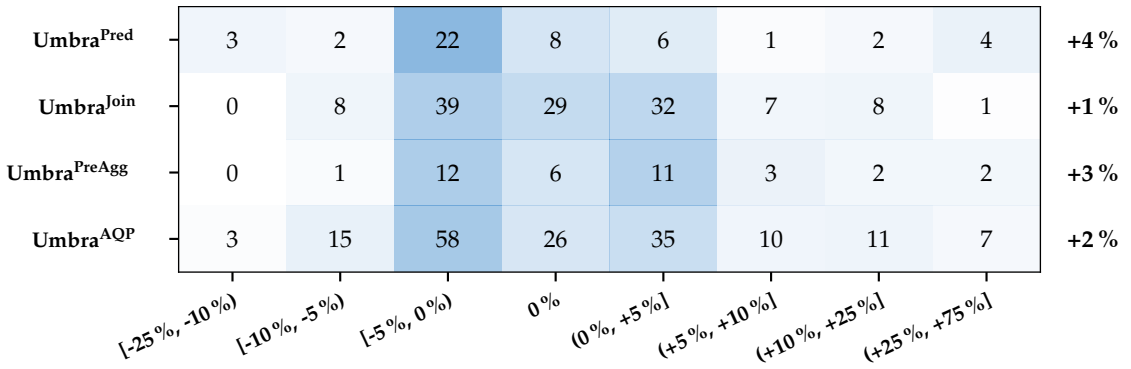


Figure 7.9.: Histogram of the actual speedup of **Umbra**$^{\mathbf{AQP}}$ over **Umbra** when using our Dynamic Blocks framework, the variant-aware Flying Start compiler, and the Dynamic Execution strategy.

beneficial for statements that are difficult to estimate, for instance, *in* or *like* predicates. We observed the highest performance boost for aggregations when multiple columns are used as keys, and computing the hash value with the default implementation is expensive. Most of the queries use the Dynamic Join optimization, and several minor improvements can be explained with fluctuations in the execution time. Nevertheless, we measure for several queries a clear speedup if Umbra's query optimizer was not able to find the optimal join order.

The second part of the question, on the other hand, is more ambivalent. While overall, we gain performance, the boost is not as big as theoretically possible. We lose roughly 2 % of the speedup due to the compilation and execution overhead in **Umbra**$^{\mathbf{AQP}}$ compared to **Umbra**. As a result, 46 % of the queries are slower with Adaptive Query Processing. Although most of these queries exhibit only a minor slowdown, for 10 % of the queries, the performance drops by more than 5 %. The slowdown is mostly caused by the fluctuations in the execution times of the generated programs (cf. Section 7.3) and exploring suboptimal variations. However, in some cases, the fluctuations in the execution can also improve the runtime: a few queries with speedup between 1 % and 25 % are faster even though they use the default implementation.

8. Discussion and Future Work

The following chapter discusses some of the decisions we made then implementing Adaptive Query Processing in Umbra and additional work for the future. Although the existing dynamic optimizations already provide significant performance improvements in Umbra, query processing can be improved further. We believe that our existing Dynamic Blocks framework is the perfect building block for these enhancements. Nonetheless, the Dynamic Execution strategy needs additional refinements to react to changes in the processed data and combine it with the existing LLVM-based backend. The following improvements are possible for the three components of the Adaptive Query Processing in Umbra.

Dynamic Blocks framework (Chapter 4) The three dynamic block types we proposed in the framework can represent the adaptive optimization from NoisePage and integrate well into the existing Tidy Tuples framework. However, more block types might be necessary for further adapting the code and avoiding recompilations. For instance, we could use an optional-like block for instrumenting the generated code and collect runtime statistics on samples without generating the code twice. Furthermore, we plan to implement more dynamic optimizations to adapt the query plan and optimize expression evaluation. In addition, the existing dynamic optimization and optimizer that decides whether to apply them require further fine-tuning to avoid performance regressions.

Variant-aware Flying Start compiler (Chapter 5) We modified Umbra's Flying Start compiler to generate machine code for the variations fast and with no execution overhead. These modifications make it possible to lower dynamic blocks to machine code without recompilation by rearranging the generated instructions. While compilation time increases minimally, we observed overall no relevant slowdown when executing the rewritten code. Nevertheless, it might be beneficial to compile pipelines with only a few variations with the original Flying Start backend to avoid fluctuations in the execution time.

Dynamic Execution strategy (Chapter 6) Probably the most debatable decision in this thesis is that we explore the variations only once in the beginning. The reason for this design decision is the future goal to compile the best-performing variation with

the LLVM-based backend after exploring all variations. This last step will allow us to combine the benefits of Adaptive Query Execution with Adaptive Execution, which is already available in Umbra [34]. Nevertheless, it is possible to react to changes in the processed data and choose a different variation after the initial exploration phase. We propose to re-evaluate the first initial choice if the performance of the currently exploited variation drops significantly and another variation might perform better.

Although our run-based policy for finding the best-performing variation achieves good results, a well-researched technique like Thompson sampling could perform better and shorten the exploration phase [28]. We can also reduce the length of the exploration phase by pruning the search space: When reordering joins or predicates, most of the time, only one variant changes its position. Hence, we can reduce the number of variations to explore by considering only these permutations.

Another problem arises then combining multiple dynamic optimizations in the same pipeline. Currently, we evaluate all combinations of the dynamic blocks created by the optimizations. Hence, the exploration phase grows exponentially with the number of dynamic blocks in the pipeline. This was not a problem in our experiments as none of the pipelines used more than two dynamic blocks in the same pipeline. However, once more operators and expressions use our Dynamic Blocks framework, this number will increase, and the search space explodes. Again, we can solve this problem by pruning the explored variations: assuming that the dynamic blocks in a pipeline are independent of each other, we can incrementally determine the best configuration per block.

9. Conclusion

This thesis implemented Adaptive Query Processing in a compiling query engine with tuple-at-a-time processing. We propose three key techniques for efficiently generating variations for pipelines in Umbra, compiling them to machine code, and adapting the generated code during execution. Together these three components solve the initial challenges for tuple-routing adaptivity in a compiling database system: The Dynamic Blocks framework supports changes to the query's execution plan and fine-grained optimizations to the generated code. Our modified Flying Start compiler avoids recompilations of the query and the intermediate representation while emitting optimized assembly instruction. During execution, the Dynamic Execution strategy first explores different variations and then executes the best-performing one. We minimized the number of morsels used in the exploration phase, and the overhead of assembling the different variations is as small as possible.

Our experimental analysis shows performance improvements in four major benchmarks for database systems. Reordering joins, adapting the filter evaluation, and optimizing the preaggregation improve the runtime by up to 2x. On average, we measure 2 % speedup in the 165 queries that use one of our dynamic optimizations. Although some of the queries see a minor slowdown, most of them are faster, and the maximum performance boost exceeds the losses by far. In particular, we observe similar speedups as NoisePage with the Permutable Compiled Queries [42] and can validate their findings.

In conclusion, our results show that tuple-routing Adaptive Query Processing is not only feasible in a high-performance system like Umbra but also improves the execution times. The implementation we proposed is a good starting point for future optimizations and further research in this area.

Bibliography

[1] 2021. *The LLVM Compiler Infrastructure*. https://llvm.org/

[2] 2021. *NoisePage*. https://noise.page/

[3] Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-Tuning Histograms: Building Histograms without Looking at Data. *SIGMOD Rec.* 28 (6 1999), 181–192. Issue 2. https://doi.org/10.1145/304181.304198

[4] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. *SIGMOD Rec.* 29 (5 2000), 261–272. Issue 2. https://doi.org/10.1145/335191.335420

[5] Shivnath Babu and Pedro Bizarro. 2005. Adaptive Query Processing in the Looking Glass. *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, 238–249. http://cidrdb.org/cidr2005/papers/P20.pdf

[6] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive Re-Optimization. *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 107–118. https://doi.org/10.1145/1066157.1066171

[7] Pedro Bizarro, Nicolas Bruno, and David J DeWitt. 2009. Progressive Parametric Query Optimization. *IEEE Trans. Knowl. Data Eng.* 21 (2009), 582–594. Issue 4. https://doi.org/10.1109/TKDE.2008.160

[8] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[9] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. 2008. A Pay-as-You-Go Framework for Query Execution Feedback. *Proc. VLDB Endow.* 1 (8 2008), 1141–1152. Issue 1. https://doi.org/10.14778/1453856.1453977

[10] Chungmin Melvin Chen and Nick Roussopoulos. 1994. Adaptive Selectivity Estimation Using Query Feedback. *SIGMOD Rec.* 23 (5 1994), 161–172. Issue 2. https://doi.org/10.1145/191843.191874

[11] Richard L Cole and Goetz Graefe. 1994. Optimization of Dynamic Query Evaluation Plans. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 150–160. https://doi.org/10.1145/191839.191872

[12] Amol Deshpande and Joseph M Hellerstein. 2004. Lifting the Burden of History from Adaptive Query Processing. *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, 948–959.

[13] A Deshpande, J M Hellerstein, and V Raman. 2006. Adaptive Query Processing: Why, How, When, What Next. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 806–807. https://doi.org/10.1145/1142473.1142603

[14] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. *Foundations and Trends in Databases* 1 (2007), 1–140. Issue 1. https://doi.org/10.1561/1900000001

[15] Anshuman Dutt and Jayant R Haritsa. 2014. Plan Bouquets: Query Processing without Selectivity Estimation. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 1039–1050. https://doi.org/10.1145/2588555.2588566

[16] Grégory M Essertel, Ruby Y Tahboub, James M Decker, Kevin J Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-up Architectures and Medium-Size Data. *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, 799–815.

[17] Agner Fog. 2021. *The microarchitecture of Intel, AMD, and VIA CPUs - An optimization guide for assembly programmers and compiler makers*. https://www.agner.org/optimize/microarchitecture.pdf

[18] Goetz Graefe and William J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. *Proceedings of IEEE 9th International Conference on Data Engineering*, 209–218. https://doi.org/10.1109/ICDE.1993.344061

[19] Goetz Graefe and Karen Ward. 1989. Dynamic Query Evaluation Plans, James Clifford, Bruce G Lindsay, and David Maier (Eds.). *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, 358–366. https://doi.org/10.1145/67544.66960

[20] Philipp M Grulich, Breß{} Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2487–2503. https://doi.org/10.1145/3318464.3389739

[21] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1917–1923. https://doi.org/10.1145/2723372.2742795

[22] Matthew S Hecht and Jeffrey D Ullman. 1972. Flow Graph Reducibility. *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, 238–250. https://doi.org/10.1145/800152.804919

[23] Joseph M Hellerstein, Michael J Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A Shah. 2000. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin* 23 (2000), 12.

[24] Arvind Hulgeri and S. Sudarshan. 2002. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, 167–178. https://doi.org/10.1016/B978-155860869-6/50023-8

[25] Arvind Hulgeri and S. Sudarshan. 2003. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions, Johann Christoph Freytag, Peter C Lockemann, Serge Abiteboul, Michael J Carey, Patricia G Selinger, and Andreas Heuer (Eds.). *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, 766–777. https://doi.org/10.1016/B978-012722442-8/50073-2

[26] Zachary G Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S Weld. 1999. An Adaptive Query Execution System for Data Integration. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 299–310. https://doi.org/10.1145/304182.304209

[27] Navin Kabra and David J DeWitt. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 106–117. https://doi.org/10.1145/276304.276315

[28] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *CoRR* abs/1802.0 (2018). http://arxiv.org/abs/1802.09180

[29] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[30] Timo Kersten. 2021. Optimizing Relational Query Engine Architecture for Modern Hardware.

[31] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11 (9 2018), 2209–2222. Issue 13.

[32] 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 (2021).

[33] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 197–208. https://doi.org/10.1109/ICDE.2018.00027

[34] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Making Compiling Query Engines Practical. *IEEE Transactions on Knowledge and Data Engineering* 33 (2 2021), 597–612. Issue 2. https://doi.org/10.1109/TKDE.2019.2905235

[35] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[36] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 743–754. https://doi.org/10.1145/2588555.2610507

[37] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9 (11 2015), 204–215. Issue 3. https://doi.org/10.14778/2850583.2850594

[38] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query Optimization through the Looking Glass, and What We Found Running the Join Order Benchmark. *The VLDB Journal* 27 (10 2018), 643–668. Issue 5. https://doi.org/10.1007/s00778-017-0480-7

[39] Quanzhong Li, Minglong Shao, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. 2007. Adaptively Reordering Joins during Query Execution. *2007 IEEE 23rd International Conference on Data Engineering*, 26–35. https://doi.org/10.1109/ICDE.2007.367848

[40] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust Query Processing through Progressive Optimization. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 659–670. https://doi.org/10.1145/1007568.1007642

[41] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *Proc. VLDB Endow.* 11 (9 2017), 1–13. Issue 1. https://doi.org/10.14778/3151113.3151114

[42] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C Mowry, and Andrew Pavlo. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *Proc. VLDB Endow.* 14 (10 2020), 101–113. Issue 2. https://doi.org/10.14778/3425879.3425882

[43] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4 (6 2011), 539–550. Issue 9. https://doi.org/10.14778/2002938.2002940

[44] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. *CIDR* (2020).

[45] Thomas Neumann and César A Galindo-Legaria. 2013. Taking the Edge off Cardinality Estimation Errors using Incremental Execution, Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen (Eds.). *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings* P-214, 73–92. https://dl.gi.de/20.500.12116/17356

[46] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.). *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings* P-241, 383–402. https://dl.gi.de/20.500.12116/2418

[47] Kenneth W Ng, Zhenghao Wang, Richard R Muntz, and Silvia Nittel. 1999. Dynamic Query Re-Optimization, Z Meral Özsoyoglu, Gultekin Özsoyoglu, and Wen-Chi Hou (Eds.). *11th International Conference on Scientific and Statistical Database Management, Proceedings, Cleveland, Ohio, USA, 28-30 July, 1999*, 264–273. https://doi.org/10.1109/SSDM.1999.787642

[48] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, 1758–1761. https://doi.org/10.1109/ICDE.2019.00191

[49] Vijayshankar Raman, Amol Deshpande, and Joseph M Hellerstein. 2003. Using State Modules for Adaptive Query Processing, Umeshwar Dayal, Krithi Ramamritham, and T. M Vijayaraman (Eds.). *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, 353–364. https://doi.org/10.1109/ICDE.2003.1260805

[50] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 12–27. https://doi.org/10.1145/73560.73562

[51] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro Adaptivity in Vectorwise. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 1231–1242. https://doi.org/10.1145/2463676.2465292

[52] Patricia G Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access Path Selection in a Relational Database Management System. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 23–34. https://doi.org/10.1145/582095.582099

[53] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. *Proceedings of the 27th International Conference on Very Large Data Bases*, 19–28.

[54] Robert Tarjan. 1973. Testing Flow Graph Reducibility. *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, 96–107. https://doi.org/10.1145/800125.804040

[55] W R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25 (1933), 285–294. Issue 3–4.

[56] Feng Tian and David J DeWitt. 2003. Tuple Routing Strategies for Distributed Eddies. *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, 333–344.

[57] Joannès Vermorel and Mehryar Mohri. 2005. Multi-armed Bandit Algorithms and Empirical Evaluation, João Gama, Rui Camacho, Pavel Brazdil, Al{ı}pio Jorge, and Lu{ı}s Torgo (Eds.). *Machine Learning: ECML 2005, 16th European Conference*

on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings 3720, 437–448. https://doi.org/10.1007/11564096_42

[58] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. 2020. Meet Me Halfway: Split Maintenance of Continuous Views. *Proc. VLDB Endow.* 13 (7 2020), 2620–2633. Issue 12. https://doi.org/10.14778/3407790.3407849

[59] Eugene Wong and Karel Youssefi. 1976. Decomposition—a Strategy for Query Processing. *ACM Trans. Database Syst.* 1 (9 1976), 223–241. Issue 3. https://doi.org/10.1145/320473.320479

[60] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. 2016. Non-Invasive Progressive Optimization for in-Memory Databases. *Proc. VLDB Endow.* 9 (10 2016), 1659–1670. Issue 14. https://doi.org/10.14778/3007328.3007332

[61] Marcin Zukowski and Peter A Boncz. 2012. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.* 35 (2012), 21–27. Issue 1. http://sites.computer.org/debull/A12mar/vectorwise.pdf

Appendix

A. Runtimes

In total, 165 of the 251 queries in the four benchmarks use at least one dynamic optimization. We report for all queries the original runtime using the Flying Start compiler without dynamic blocks as backend (**Umbra**). Furthermore, the speedup and execution time for Dynamic Predicates (**Umbra^{Pred}**), Dynamic Joins (**Umbra^{Join}**), or Dynamic Preaggregation (**Umbra^{PreAgg}**) are shown. **Umbra^{AQP}** combines all three optimizations and uses the Flying Start backend with dynamic blocks.

Table A.1.: Runtimes and speedups for TPC-H queries that use a dynamic optimization.

Query	Umbra	Umbra^{Pred}		Umbra^{Join}		Umbra^{PreAgg}		Umbra^{AQP}	
Q1	151.3 ms	-		-		158.8 ms	-5 %	161.3 ms	-6 %
Q2	10.6 ms	11.2 ms	-5 %	8.0 ms	+33 %	-		8.1 ms	+31 %
Q3	69.6 ms	-		-		69.6 ms	0 %	69.5 ms	0 %
Q5	48.0 ms	-		-		48.6 ms	-1 %	48.6 ms	-1 %
Q6	25.6 ms	24.5 ms	+4 %	-		24.8 ms	+3 %	24.7 ms	+4 %
Q7	56.9 ms	-		60.2 ms	-5 %	-		59.5 ms	-4 %
Q8	44.3 ms	-		45.3 ms	-2 %	-		44.6 ms	-1 %
Q9	195.5 ms	-		-		197.4 ms	-1 %	197.2 ms	-1 %
Q10	67.8 ms	-		-		68.0 ms	0 %	68.1 ms	-1 %
Q12	46.8 ms	42.5 ms	+10 %	-		47.1 ms	-1 %	42.0 ms	+11 %
Q13	99.3 ms	-		-		87.8 ms	+13 %	87.9 ms	+13 %
Q14	30.9 ms	-		-		31.0 ms	0 %	31.0 ms	0 %
Q15	26.8 ms	-		-		26.1 ms	+3 %	26.1 ms	+3 %
Q16	56.9 ms	56.9 ms	0 %	-		55.3 ms	+3 %	56.3 ms	+1 %
Q17	363.4 ms	312.3 ms	+16 %	-		308.2 ms	+18 %	310.8 ms	+17 %
Q18	179.5 ms	-		-		167.0 ms	+8 %	166.0 ms	+8 %
Q19	84.0 ms	86.2 ms	-2 %	-		-		86.2 ms	-2 %
Q20	29.8 ms	-		32.2 ms	-7 %	-		32.1 ms	-7 %
Q22	49.9 ms	50.3 ms	-1 %	-		50.7 ms	-2 %	50.2 ms	-1 %

Table A.2.: Runtimes and speedups for SSB queries that use a dynamic optimization.

Query	Umbra	Umbra^{Pred}		Umbra^{Join}		Umbra^{PreAgg}		Umbra^{AQP}	
Q11	43.7 ms	31.3 ms	+39 %	-		41.9 ms	+4 %	31.5 ms	+39 %
Q21	50.3 ms	-		50.2 ms	0 %	51.5 ms	-2 %	50.6 ms	-1 %
Q22	36.5 ms	-		32.9 ms	+11 %	-		32.9 ms	+11 %
Q23	31.2 ms	-		26.4 ms	+18 %	-		26.4 ms	+18 %
Q31	69.4 ms	-		69.5 ms	0 %	69.4 ms	0 %	72.4 ms	-4 %

Query	Umbra	Umbra^{Pred}		Umbra^{Join}		Umbra^{PreAgg}		Umbra^{AQP}	
Q41	77.4 ms	-		78.2 ms	-1 %	79.9 ms	-3 %	82.1 ms	-6 %
Q42	71.0 ms	-		64.2 ms	+11 %	-		64.0 ms	+11 %

Table A.3.: Runtimes and speedups for TPC-DS queries that use a dynamic optimization.

Query	Umbra	Umbra^{Pred}		Umbra^{Join}		Umbra^{PreAgg}		Umbra^{AQP}	
Q2	61.3 ms	-		-		59.7 ms	+3 %	59.6 ms	+3 %
Q4	762.7 ms	-		-		604.7 ms	+26 %	604.6 ms	+26 %
Q5	66.7 ms	-		-		70.5 ms	-5 %	70.4 ms	-5 %
Q7	23.0 ms	23.4 ms	-1 %	20.8 ms	+11 %	-		20.7 ms	+11 %
Q8	20.8 ms	-		19.5 ms	+7 %	20.6 ms	+1 %	19.1 ms	+9 %
Q9	156.5 ms	-		-		160.5 ms	-2 %	160.5 ms	-3 %
Q10	34.9 ms	-		34.2 ms	+2 %	-		34.1 ms	+2 %
Q11	420.6 ms	-		-		305.2 ms	+38 %	304.9 ms	+38 %
Q13	28.8 ms	30.8 ms	-6 %	-		-		30.8 ms	-6 %
Q14a	295.7 ms	-		-		322.8 ms	-8 %	323.6 ms	-9 %
Q14b	475.9 ms	-		-		495.3 ms	-4 %	495.0 ms	-4 %
Q16	13.5 ms	-		13.9 ms	-3 %	-		13.9 ms	-3 %
Q18	48.7 ms	48.9 ms	-1 %	-		-		48.9 ms	0 %
Q19	18.0 ms	-		16.6 ms	+9 %	-		16.5 ms	+9 %
Q21	59.1 ms	-		62.2 ms	-5 %	-		62.1 ms	-5 %
Q23a	762.9 ms	-		741.0 ms	+3 %	754.0 ms	+1 %	758.4 ms	+1 %
Q23b	768.1 ms	-		743.6 ms	+3 %	755.7 ms	+2 %	756.1 ms	+2 %
Q26	14.7 ms	14.7 ms	0 %	14.8 ms	-1 %	-		14.6 ms	+1 %
Q27	28.6 ms	28.6 ms	0 %	25.2 ms	+13 %	-		25.4 ms	+12 %
Q28	114.8 ms	115.0 ms	0 %	-		114.2 ms	+1 %	116.1 ms	-1 %
Q33	32.3 ms	-		33.3 ms	-3 %	-		33.3 ms	-3 %
Q34	18.3 ms	-		19.0 ms	-3 %	-		19.0 ms	-3 %
Q36	53.3 ms	-		53.5 ms	0 %	-		53.4 ms	0 %
Q37	73.0 ms	-		77.5 ms	-6 %	-		77.5 ms	-6 %
Q38	118.7 ms	-		-		109.9 ms	+8 %	109.8 ms	+8 %
Q40	9.0 ms	-		9.8 ms	-8 %	-		9.8 ms	-8 %
Q42	16.9 ms	-		14.7 ms	+15 %	-		14.7 ms	+15 %
Q43	32.8 ms	-		31.2 ms	+5 %	-		31.2 ms	+5 %
Q44	11.1 ms	11.8 ms	-6 %	-		-		11.9 ms	-6 %
Q46	38.1 ms	-		36.2 ms	+5 %	-		36.1 ms	+6 %
Q47	185.6 ms	-		184.9 ms	0 %	-		184.6 ms	+1 %
Q48	46.4 ms	49.0 ms	-5 %	-		-		49.1 ms	-6 %
Q52	13.9 ms	-		13.2 ms	+5 %	-		13.3 ms	+5 %
Q54	25.6 ms	-		27.6 ms	-7 %	-		27.5 ms	-7 %
Q55	14.4 ms	-		15.0 ms	-4 %	-		15.0 ms	-4 %
Q56	31.4 ms	-		28.3 ms	+11 %	-		28.6 ms	+10 %
Q59	111.2 ms	-		-		108.8 ms	+2 %	108.9 ms	+2 %
Q60	38.3 ms	-		37.3 ms	+3 %	-		37.1 ms	+3 %
Q61	13.1 ms	-		12.3 ms	+6 %	-		12.2 ms	+7 %
Q64	29.4 ms	-		30.4 ms	-4 %	-		30.6 ms	-4 %
Q66	24.8 ms	-		24.7 ms	+1 %	-		24.8 ms	0 %
Q67	1061.0 ms	-		1074.0 ms	-1 %	-		1068.2 ms	-1 %
Q68	21.8 ms	-		18.8 ms	+16 %	-		18.7 ms	+16 %
Q72	220.2 ms	-		219.4 ms	0 %	219.5 ms	0 %	218.9 ms	+1 %
Q73	16.3 ms	-		15.9 ms	+3 %	-		15.9 ms	+3 %

Query	Umbra	Umbra^{Pred}		Umbra^{Join}		Umbra^{PreAgg}		Umbra^{AQP}	
Q74	178.4 ms	-		-		162.1 ms	+10 %	163.1 ms	+9 %
Q75	165.8 ms	-		165.3 ms	0 %	-		165.3 ms	0 %
Q79	20.6 ms	-		20.7 ms	-1 %	-		20.7 ms	-1 %
Q80	48.0 ms	-		47.2 ms	+2 %	-		47.0 ms	+2 %
Q81	14.7 ms	-		15.3 ms	-4 %	-		15.1 ms	-3 %
Q82	71.1 ms	-		78.1 ms	-9 %	-		78.2 ms	-9 %
Q84	3.6 ms	-		3.6 ms	0 %	-		3.6 ms	0 %
Q85	11.8 ms	12.2 ms	-3 %	-		-		11.9 ms	0 %
Q87	118.0 ms	-		-		115.5 ms	+2 %	115.3 ms	+2 %
Q88	115.4 ms	-		110.7 ms	+4 %	-		110.9 ms	+4 %
Q89	34.5 ms	-		34.6 ms	0 %	-		34.6 ms	0 %
Q94	9.1 ms	-		8.4 ms	+8 %	-		8.4 ms	+9 %
Q95	159.0 ms	-		159.2 ms	0 %	-		158.6 ms	0 %
Q96	14.0 ms	-		13.5 ms	+4 %	-		13.6 ms	+3 %
Q98	36.0 ms	-		34.5 ms	+4 %	-		34.3 ms	+5 %

Table A.4.: Runtimes and speedups for JOB queries that use a dynamic optimization.

Query	Umbra	Umbra^{Pred}		Umbra^{Join}		Umbra^{PreAgg}		Umbra^{AQP}	
Q1a	6.5 ms	6.7 ms	-3 %	6.6 ms	-3 %	-		6.6 ms	-3 %
Q1b	3.9 ms	-		4.0 ms	0 %	-		4.0 ms	0 %
Q1c	5.5 ms	5.6 ms	-2 %	5.5 ms	0 %	-		5.6 ms	-2 %
Q1d	3.8 ms	-		3.9 ms	-3 %	-		3.9 ms	-3 %
Q2b	6.7 ms	-		6.6 ms	+2 %	-		6.7 ms	+1 %
Q2c	4.2 ms	-		4.3 ms	-2 %	-		4.3 ms	-3 %
Q5a	14.5 ms	14.8 ms	-2 %	14.5 ms	0 %	-		14.7 ms	-1 %
Q5b	19.5 ms	19.9 ms	-2 %	19.7 ms	-1 %	-		20.0 ms	-3 %
Q5c	23.3 ms	24.3 ms	-4 %	21.8 ms	+7 %	-		22.5 ms	+4 %
Q6c	16.6 ms	-		16.5 ms	0 %	-		16.5 ms	0 %
Q8a	18.1 ms	19.1 ms	-5 %	-		-		19.1 ms	-5 %
Q8b	21.8 ms	19.1 ms	+14 %	-		-		19.1 ms	+14 %
Q8c	110.0 ms	-		105.8 ms	+4 %	110.7 ms	-1 %	105.2 ms	+5 %
Q8d	37.0 ms	-		36.5 ms	+1 %	37.7 ms	-2 %	36.4 ms	+2 %
Q9a	52.8 ms	53.5 ms	-1 %	-		-		53.6 ms	-1 %
Q9b	25.8 ms	27.2 ms	-5 %	26.8 ms	-4 %	-		27.1 ms	-5 %
Q10a	31.5 ms	32.6 ms	-3 %	30.7 ms	+3 %	-		32.9 ms	-4 %
Q10b	30.8 ms	-		29.9 ms	+3 %	-		30.0 ms	+3 %
Q10c	62.1 ms	-		-		62.1 ms	0 %	62.7 ms	-1 %
Q11a	4.9 ms	-		4.9 ms	+1 %	-		4.9 ms	-1 %
Q11b	5.6 ms	5.8 ms	-4 %	5.1 ms	+9 %	-		5.6 ms	0 %
Q11c	9.0 ms	-		9.0 ms	+1 %	-		9.2 ms	-1 %
Q11d	8.8 ms	-		8.9 ms	-1 %	-		8.9 ms	-1 %
Q12a	14.9 ms	-		15.4 ms	-3 %	-		15.3 ms	-3 %
Q12b	12.7 ms	12.6 ms	+1 %	12.8 ms	-1 %	-		13.5 ms	-6 %
Q12c	17.5 ms	-		17.9 ms	-3 %	-		18.0 ms	-3 %
Q13a	38.6 ms	-		38.5 ms	0 %	-		38.4 ms	+1 %
Q13b	19.2 ms	19.2 ms	0 %	21.2 ms	-10 %	-		21.1 ms	-9 %
Q13c	16.9 ms	17.1 ms	-2 %	18.4 ms	-8 %	-		18.6 ms	-9 %
Q13d	57.5 ms	-		52.8 ms	+9 %	-		52.3 ms	+10 %
Q14a	34.8 ms	-		36.0 ms	-3 %	-		35.2 ms	-1 %
Q14b	34.3 ms	-		34.5 ms	-1 %	-		34.5 ms	-1 %

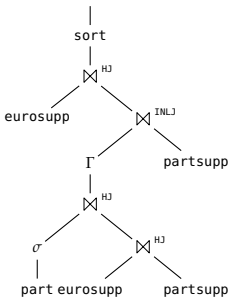
Query	Umbra	Umbra^{Pred}		Umbra^{Join}		Umbra^{PreAgg}	Umbra^{AQP}	
Q14c	36.8 ms	-		36.9 ms	0 %	-	37.1 ms	-1 %
Q15a	24.7 ms	14.5 ms	+71 %	-		-	14.4 ms	+72 %
Q15b	25.0 ms	14.9 ms	+68 %	25.0 ms	0 %	-	14.9 ms	+67 %
Q15c	12.5 ms	15.9 ms	-22 %	-		-	15.8 ms	-21 %
Q18a	116.8 ms	-		116.1 ms	+1 %	-	116.4 ms	0 %
Q18b	44.5 ms	43.9 ms	+1 %	44.3 ms	0 %	-	44.0 ms	+1 %
Q18c	52.5 ms	-		52.6 ms	0 %	-	52.9 ms	-1 %
Q19a	81.4 ms	81.8 ms	0 %	81.6 ms	0 %	-	82.1 ms	-1 %
Q19b	48.2 ms	48.4 ms	0 %	48.2 ms	0 %	-	48.7 ms	-1 %
Q19c	79.3 ms	-		79.2 ms	0 %	-	79.5 ms	0 %
Q19d	279.6 ms	-		276.5 ms	+1 %	-	277.1 ms	+1 %
Q21a	23.0 ms	-		22.8 ms	+1 %	-	22.9 ms	0 %
Q21b	15.3 ms	-		14.9 ms	+2 %	-	14.8 ms	+3 %
Q21c	34.1 ms	-		33.9 ms	0 %	-	34.0 ms	0 %
Q22a	33.2 ms	33.1 ms	+1 %	33.9 ms	-2 %	-	33.6 ms	-1 %
Q22b	32.7 ms	33.0 ms	-1 %	33.6 ms	-3 %	-	33.2 ms	-1 %
Q22c	39.8 ms	39.2 ms	+2 %	38.9 ms	+2 %	-	38.9 ms	+2 %
Q22d	43.6 ms	-		42.9 ms	+1 %	-	43.1 ms	+1 %
Q23a	13.4 ms	16.5 ms	-19 %	-		-	16.6 ms	-19 %
Q23b	23.4 ms	13.5 ms	+73 %	23.4 ms	0 %	-	13.4 ms	+74 %
Q23c	13.5 ms	16.3 ms	-17 %	-		-	16.4 ms	-18 %
Q24a	81.2 ms	-		81.6 ms	-1 %	-	81.7 ms	-1 %
Q24b	81.0 ms	-		81.1 ms	0 %	-	80.5 ms	+1 %
Q25a	49.8 ms	-		50.1 ms	-1 %	-	50.0 ms	0 %
Q25b	48.7 ms	-		49.8 ms	-2 %	-	49.1 ms	-1 %
Q25c	53.1 ms	-		53.4 ms	0 %	-	53.4 ms	0 %
Q26a	21.9 ms	-		23.5 ms	-7 %	-	23.5 ms	-7 %
Q26b	20.1 ms	-		19.9 ms	+1 %	-	20.1 ms	0 %
Q26c	24.3 ms	-		24.1 ms	+1 %	-	24.1 ms	+1 %
Q27a	17.0 ms	-		17.1 ms	0 %	-	17.1 ms	0 %
Q27b	16.2 ms	-		16.3 ms	-1 %	-	16.5 ms	-2 %
Q27c	27.1 ms	-		27.2 ms	0 %	-	27.4 ms	-1 %
Q28a	31.7 ms	32.7 ms	-3 %	31.4 ms	+1 %	-	31.9 ms	-1 %
Q28b	22.9 ms	23.4 ms	-2 %	23.3 ms	-2 %	-	23.4 ms	-2 %
Q28c	33.5 ms	34.0 ms	-1 %	33.0 ms	+2 %	-	33.1 ms	+1 %
Q29a	83.8 ms	83.4 ms	0 %	84.4 ms	-1 %	-	84.0 ms	0 %
Q29b	65.0 ms	64.0 ms	+1 %	64.8 ms	0 %	-	64.6 ms	0 %
Q29c	139.7 ms	-		139.7 ms	0 %	-	140.1 ms	0 %
Q30a	48.9 ms	-		49.4 ms	-1 %	-	49.5 ms	-1 %
Q30b	48.7 ms	-		49.2 ms	-1 %	-	49.0 ms	-1 %
Q30c	50.8 ms	-		51.6 ms	-2 %	-	51.4 ms	-1 %
Q31a	56.8 ms	-		57.2 ms	-1 %	-	56.9 ms	0 %
Q31b	50.9 ms	-		51.2 ms	-1 %	-	51.3 ms	-1 %
Q31c	81.9 ms	-		82.3 ms	-1 %	-	82.3 ms	-1 %
Q33a	9.2 ms	-		9.1 ms	+1 %	-	9.2 ms	0 %
Q33b	8.8 ms	-		8.8 ms	-1 %	-	8.8 ms	-1 %
Q33c	8.5 ms	-		8.4 ms	+2 %	-	8.4 ms	+1 %

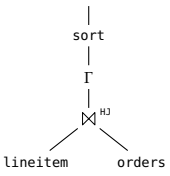
B. Query Plans

Section 7.4 inspected several queries from the four benchmarks (TPC-H, SSB, TPC-DS, and JOB) closer. The following tables contain the SQL statements and the corresponding query execution plan used by Umbra. For some of the queries, we only show the relevant part of the plan where the dynamic optimizations are applied. The full plans can be found in the Umbra web interface (`https://umbra-db.com/interface/`). Please note that the plans shown in this thesis might differ from the ones shown online due to updates to Umbra's query optimizer.

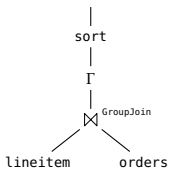
Table B.1.: SQL statements and query plans for the four queries from TPC-H.

	SQL statement	Query Plan

Q2	`with eurosupp as (select s.*`
	`  from supplier s, nation, region`
	`  where s.s_nationkey = n_nationkey`
	`  and n_regionkey = r_regionkey`
	`  and r_name = 'EUROPE')`
	`select s_acctbal, s_name, n_name, p_partkey, p_mfgr,`
	`  s_address, s_phone, s_comment`
	`from part, eurosupp, partsupp,`
	`where p_partkey = ps_partkey and s_suppkey = ps_suppkey`
	`  and p_size = 15 and p_type like '%BRASS'`
	`  and ps_supplycost = (select min(ps_supplycost)`
	`  from partsupp, eurosupp`
	`  where p_partkey = ps_partkey`
	`  and s_suppkey = ps_suppkey)`
	`order by s_acctbal desc, n_name, s_name, p_partkey`
	`limit 100;`
		

Q12	`select l_shipmode,`
	`  sum(case when o_orderpriority = '1-URGENT'`
	`  or o_orderpriority = '2-HIGH'`
	`  then 1 else 0 end) as high_line_count,`
	`  sum(case when o_orderpriority <> '1-URGENT'`
	`  and o_orderpriority <> '2-HIGH'`
	`  then 1 else 0 end) as low_line_count`
	`from orders, lineitem`
	`where o_orderkey = l_orderkey`
	`  and l_shipmode in ('MAIL', 'SHIP')`
	`  and l_commitdate < l_receiptdate`
	`  and l_shipdate < l_commitdate`
	`  and l_receiptdate >= date '1994-01-01'`
	`  and l_receiptdate < date '1994-01-01' + interval '1' year`
	`group by l_shipmode`
	`order by l_shipmode;`
		

	SQL statement	Query Plan

Q13	`select c_count, count(*) as custdist`
	`from (select c_custkey, count(o_orderkey)`
	`  from customer left outer join orders on`
	`  c_custkey = o_custkey`
	`  and o_comment not like '%special%requests%'`
	`  group by c_custkey) as c_orders (c_custkey, c_count)`
	`group by c_count`
	`order by custdist desc, c_count desc;`
		

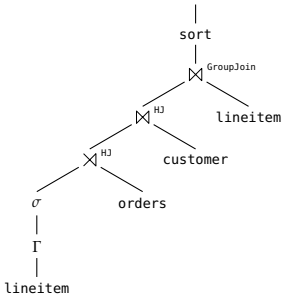
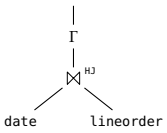
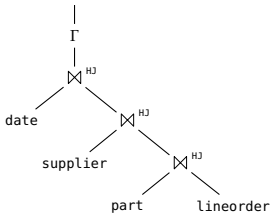
Q18	`select c_name, c_custkey, o_orderkey, o_orderdate,`
	`  o_totalprice, sum(l_quantity)`
	`from customer, orders, lineitem`
	`where o_orderkey in (select l_orderkey from lineitem`
	`  group by l_orderkey having sum(l_quantity) > 300)`
	`  and c_custkey = o_custkey and o_orderkey = l_orderkey`
	`group by c_name, c_custkey, o_orderkey,`
	`  o_orderdate, o_totalprice`
	`order by o_totalprice desc, o_orderdate`
	`limit 100;`
		

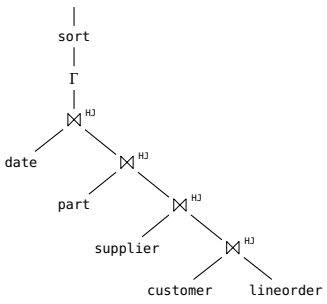
Table B.2.: SQL statements and query plans for the four queries from SSB.

	SQL statement	Query Plan

Q11	`select sum(lo_extendedprice*lo_discount) as revenue`
	`from lineorder, date`
	`where lo_orderdate = d_datekey and d_year = 1993`
	`  and lo_discount between 1 and 3 and lo_quantity < 25;`
		

Q22	`select sum(lo_revenue), d_year, p_brand1`
	`from lineorder, date, part, supplier`
	`  where lo_orderdate = d_datekey and lo_partkey = p_partkey`
	`  and lo_suppkey = s_suppkey and p_brand1 between 'MFGR#2221'`
	`  and 'MFGR#2228' and s_region = 'ASIA'`
	`group by d_year, p_brand1`
	`order by d_year, p_brand1;`
		

	SQL statement	Query Plan

Q41	**select** d_year, c_nation,
	  sum(lo_revenue - lo_supplycost) **as** profit
	from date, customer, supplier, part, lineorder
	where lo_custkey = c_custkey **and** lo_suppkey = s_suppkey
	  and lo_partkey = p_partkey **and** lo_orderdate = d_datekey
	  and c_region = 'AMERICA' **and** s_region = 'AMERICA'
	  and (p_mfgr = 'MFGR#1' **or** p_mfgr = 'MFGR#2')
	group by d_year, c_nation
	order by d_year, c_nation;
		

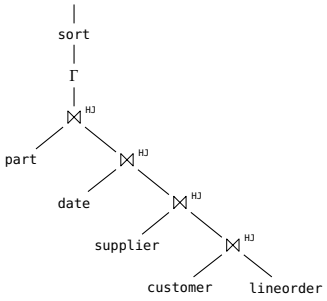
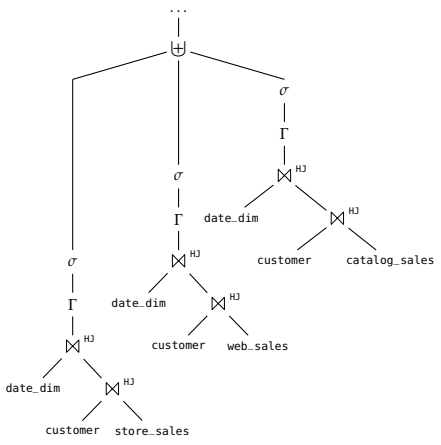
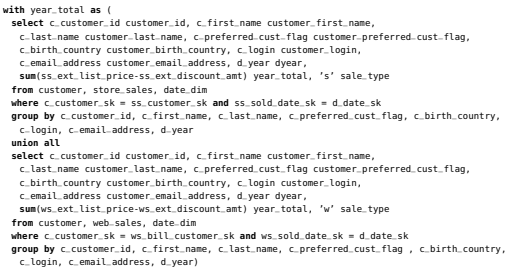
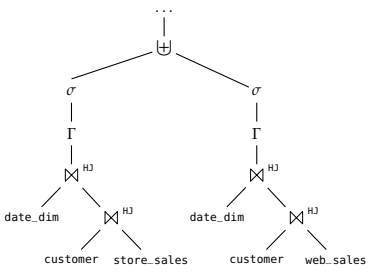
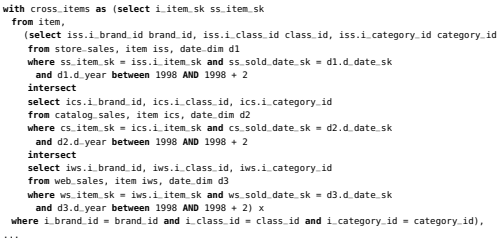
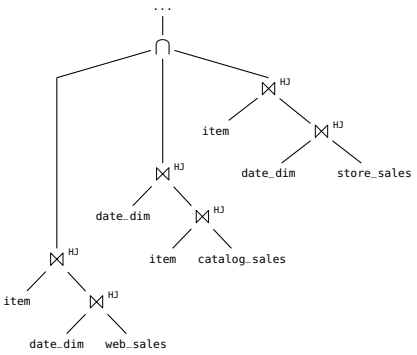
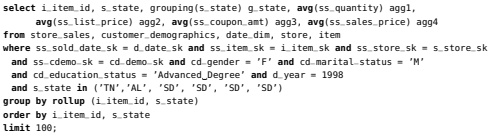
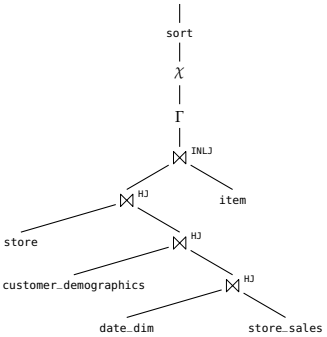
Q42	**select** d_year, s_nation, p_category,
	  sum(lo_revenue - lo_supplycost) **as** profit
	from date, customer, supplier, part, lineorder
	where lo_custkey = c_custkey **and** lo_suppkey = s_suppkey
	  and lo_partkey = p_partkey **and** lo_orderdate = d_datekey
	  and c_region = 'AMERICA' **and** s_region = 'AMERICA'
	  and (d_year = 1997 **or** d_year = 1998)
	  and (p_mfgr = 'MFGR#1' **or** p_mfgr = 'MFGR#2')
	group by d_year, s_nation, p_category
	order by d_year, s_nation, p_category;
		

Table B.3.: SQL statements and query plans for the four queries from TPC-DS.

	SQL statement	Query Plan

Q4	**with** year_total **as** (
	  select c_customer_id customer_id, c_first_name customer_first_name,
	  c_last_name customer_last_name, c_preferred_cust_flag customer_preferred_cust_flag,
	  c_birth_country customer_birth_country, c_login customer_login,
	  c_email_address customer_email_address, d_year dyear,
	  sum(((ss_ext_list_price-ss_ext_wholesale_cost-ss_ext_discount_amt)+ss_ext_sales_price)/2)
	  year_total, 's' sale_type
	  from customer, store_sales, date_dim
	  where c_customer_sk = ss_customer_sk **and** ss_sold_date_sk = d_date_sk
	  group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country,
	  c_login, c_email_address, d_year
	  union all
	  select c_customer_id customer_id, c_first_name customer_first_name,
	  c_last_name customer_last_name, c_preferred_cust_flag customer_preferred_cust_flag,
	  c_birth_country customer_birth_country, c_login customer_login,
	  c_email_address customer_email_address, d_year dyear,
	  sum((((cs_ext_list_price-cs_ext_wholesale_cost-cs_ext_discount_amt)+cs_ext_sales_price)/2))
	  year_total, 'c' sale_type
	  from customer, catalog_sales, date_dim
	  where c_customer_sk = cs_bill_customer_sk **and** cs_sold_date_sk = d_date_sk
	  group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country,
	  c_login, c_email_address, d_year
	  union all
	  select c_customer_id customer_id, c_first_name customer_first_name,
	  c_last_name customer_last_name, c_preferred_cust_flag customer_preferred_cust_flag,
	  c_birth_country customer_birth_country, c_login customer_login,
	  c_email_address customer_email_address, d_year dyear,
	  sum(((ws_ext_list_price-ws_ext_wholesale_cost-ws_ext_discount_amt)+ws_ext_sales_price)/2))
	  year_total, 'w' sale_type
	  from customer, web_sales, date_dim
	  where c_customer_sk = ws_bill_customer_sk **and** ws_sold_date_sk = d_date_sk
	  group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country,
	  c_login, c_email_address, d_year)
	...
		

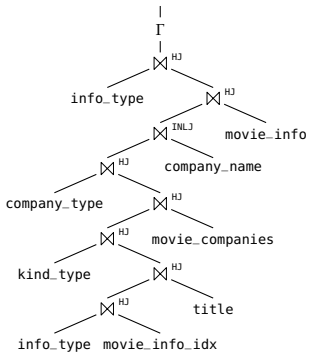
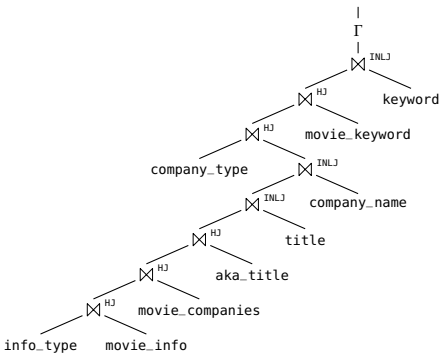
	SQL statement	Query Plan
Q11		
Q14a		
Q27		


```
with year_total as (
  select c_customer_id customer_id, c_first_name customer_first_name,
    c_last_name customer_last_name, c_preferred_cust_flag customer_preferred_cust_flag,
    c_birth_country customer_birth_country, c_login customer_login,
    c_email_address customer_email_address, d_year dyear,
    sum(ss_ext_list_price-ss_ext_discount_amt) year_total, 's' sale_type
  from customer, store_sales, date_dim
  where c_customer_sk = ss_customer_sk and ss_sold_date_sk = d_date_sk
  group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country,
    c_login, c_email_address, d_year
  union all
  select c_customer_id customer_id, c_first_name customer_first_name,
    c_last_name customer_last_name, c_preferred_cust_flag customer_preferred_cust_flag,
    c_birth_country customer_birth_country, c_login customer_login,
    c_email_address customer_email_address, d_year dyear,
    sum(ws_ext_list_price-ws_ext_discount_amt) year_total, 'w' sale_type
  from customer, web_sales, date_dim
  where c_customer_sk = ws_bill_customer_sk and ws_sold_date_sk = d_date_sk
  group by c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag , c_birth_country,
    c_login, c_email_address, d_year)
```

```
with cross_items as (select i_item_sk ss_item_sk
  from item,
    (select iss.i_brand_id brand_id, iss.i_class_id class_id, iss.i_category_id category_id
     from store_sales, item iss, date_dim d1
     where ss_item_sk = iss.i_item_sk and ss_sold_date_sk = d1.d_date_sk
       and d1.d_year between 1998 AND 1998 + 2
     intersect
     select ics.i_brand_id, ics.i_class_id, ics.i_category_id
     from catalog_sales, item ics, date_dim d2
     where cs_item_sk = ics.i_item_sk and cs_sold_date_sk = d2.d_date_sk
       and d2.d_year between 1998 AND 1998 + 2
     intersect
     select iws.i_brand_id, iws.i_class_id, iws.i_category_id
     from web_sales, item iws, date_dim d3
     where ws_item_sk = iws.i_item_sk and ws_sold_date_sk = d3.d_date_sk
       and d3.d_year between 1998 AND 1998 + 2) x
  where i_brand_id = brand_id and i_class_id = class_id and i_category_id = category_id),
...
```

```
select i_item_id, s_state, grouping(s_state) g_state, avg(ss_quantity) agg1,
       avg(ss_list_price) agg2, avg(ss_coupon_amt) agg3, avg(ss_sales_price) agg4
from store_sales, customer_demographics, date_dim, store, item
where ss_sold_date_sk = d_date_sk and ss_item_sk = i_item_sk and ss_store_sk = s_store_sk
  and ss_cdemo_sk = cd_demo_sk and cd_gender = 'F' and cd_marital_status = 'M'
  and cd_education_status = 'Advanced_Degree' and d_year = 1998
  and s_state in ('TN','AL', 'SD', 'SD', 'SD', 'SD')
group by rollup (i_item_id, s_state)
order by i_item_id, s_state
limit 100;
```

Table B.4.: SQL statements and query plans for the four queries from JOB.

	SQL statement	Query Plan
Q13b	**select min**(cn.name) **as** producing_company, **min**(miidx.info) **as** rating, **min**(t.title) **as** movie_about_winning **from** company_name **as** cn, company_type **as** ct, info_type **as** it, info_type **as** it2, kind_type **as** kt, movie_companies **as** mc, movie_info **as** mi, movie_info_idx **as** miidx, title **as** t **where** cn.country_code = '[us]' **and** t.title !!= '' **and** ct.kind = 'production_companies' **and** it.info = 'rating' **and** it2.info = 'release_dates' **and** kt.kind = 'movie' **and** (t.title **like** '%Champion%' **or** t.title **like** '%Loser%') **and** mi.movie_id = t.id **and** it2.id = mi.info_type_id **and** kt.id = t.kind_id **and** mc.movie_id = t.id **and** cn.id = mc.company_id **and** ct.id = mc.company_type_id **and** miidx.movie_id = t.id **and** it.id = miidx.info_type_id **and** mi.movie_id = miidx.movie_id **and** mi.movie_id = mc.movie_id **and** miidx.movie_id = mc.movie_id;	
Q13d	**select min**(cn.name) **as** producing_company, **min**(miidx.info) **as** rating, **min**(t.title) **as** movie **from** company_name **as** cn, company_type **as** ct, info_type **as** it, info_type **as** it2, kind_type **as** kt, movie_companies **as** mc, movie_info **as** mi, movie_info_idx **as** miidx, title **as** t **where** cn.country_code = '[us]' **and** kt.kind = 'movie' **and** ct.kind = 'production_companies' **and** it.info = 'rating' **and** it2.info = 'release_dates' **and** mi.movie_id = t.id **and** it2.id = mi.info_type_id **and** kt.id = t.kind_id **and** mc.movie_id = t.id **and** cn.id = mc.company_id **and** ct.id = mc.company_type_id **and** miidx.movie_id = t.id **and** it.id = miidx.info_type_id **and** mi.movie_id = miidx.movie_id **and** mi.movie_id = mc.movie_id **and** miidx.movie_id = mc.movie_id;	
Q15a	**select min**(mi.info) **as** release_date, **min**(t.title) **as** internet_movie **from** aka_title **as** at, company_name **as** cn, company_type **as** ct, info_type **as** it1, keyword **as** k, movie_companies **as** mc, movie_info **as** mi, movie_keyword **as** mk, title **as** t **where** cn.country_code = '[us]' **and** it1.info = 'release_dates' **and** mc.note **like** '%(200%)%' **and** mc.note **like** '%(worldwide)%' **and** mi.note **like** '%internet%' **and** mi.info **like** 'USA:%_200%' **and** t.production_year > 2000 **and** t.id = at.movie_id **and** t.id = mi.movie_id **and** t.id = mk.movie_id **and** t.id = mc.movie_id **and** mk.movie_id = mi.movie_id **and** mk.movie_id = mc.movie_id **and** mk.movie_id = at.movie_id **and** mi.movie_id = mc.movie_id **and** mi.movie_id = at.movie_id **and** mc.movie_id = at.movie_id **and** k.id = mk.keyword_id **and** it1.id = mi.info_type_id **and** cn.id = mc.company_id **and** ct.id = mc.company_type_id;	
Q15c	**select min**(mi.info) **as** release_date, **min**(t.title) **as** modern_american_internet_movie **from** aka_title **as** at, company_name **as** cn, company_type **as** ct, info_type **as** it1, keyword **as** k, movie_companies **as** mc, movie_info **as** mi, movie_keyword **as** mk, title **as** t **where** cn.country_code = '[us]' **and** it1.info = 'release_dates' **and** mi.note **like** '%internet%' **and** mi.info is **not null** **and** (mi.info **like** 'USA:%_199%' **or** mi.info **like** 'USA:%_200%') **and** t.production_year > 1990 **and** t.id = at.movie_id **and** t.id = mi.movie_id **and** t.id = mk.movie_id **and** t.id = mc.movie_id **and** mk.movie_id = mi.movie_id **and** mk.movie_id = mc.movie_id **and** mk.movie_id = at.movie_id **and** mi.movie_id = mc.movie_id **and** mi.movie_id = at.movie_id **and** mc.movie_id = at.movie_id **and** k.id = mk.keyword_id **and** it1.id = mi.info_type_id **and** cn.id = mc.company_id **and** ct.id = mc.company_type_id;	