ADAPTIVE JOIN EXECUTION IN COMPILATION-BASED EXECUTION ENGINES OF DATABASES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Ankush Rayabhari

August 2022

ABSTRACT

Query compilation and adaptive query processing aim to improve the runtime and robustness of analytical databases. However, due to the high cost of compilation, standard methods for combining these involve shaping the adaptive optimization to allow reuse of a single program instead of recompiling. We combine recent developments in both of these areas to show that both compile-once and recompilation-based execution can be practical for adaptive join ordering.

We first introduce a low-latency query compilation framework that manages the trade off between compile time and execution time at all stages. First, we describe abstractions to allow easily generating intermediate representation code. Next, we detail the intermediate representation, backend and optimizations that enable similar execution performance to LLVM while achieving much faster compilation time. We then integrate online join order learning that abandons any a-priori optimization into this framework. We propose two orthogonal approaches: a compile-once approach that uses indirection to permute the join order and a recompilation approach that generates code for each join order.

We experimentally compare each approach against optimized, analytical databases (MonetDB, DuckDB) on the join order benchmark, TPC-H and JCC-H. Overall, we find that we are able to match or incur modest overheads on queries unfavorable to adaptive optimization while dominating in queries where optimizers are susceptible to choosing a disastrous query plan. We further show that our low latency compilation framework is able to improve both proposed methods across database sizes and in particular, is critical for practical recompilation-based execution.

BIOGRAPHICAL SKETCH

Ankush Rayabhari was born in Mountain View, California and grew up in Cupertino, California. He obtained his Bachelor of Science in Computer Science from Cornell University in 2020, graduating summa cum laude with honors. He previously interned at Google, working on analytical tools for data center networking and performance optimization for Gmail and Google Search.

ACKNOWLEDGEMENTS

I am incredibly grateful for the consistent support from my adviser, Immanuel Trummer. Throughout my research journey at Cornell, he gave me the opportunity to forge my own path and let me work on problems of my choice. This work would not be possible without the freedom and guidance he provided.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Query optimization typically relies on reductive data statistics such as histograms or sketches and simplifying assumptions such as uniformity and uncorrelated data. In practice, this leads to scenarios where the optimizer misses the optimal plan causing orders of magnitude increases in runtime. This has led to a plethora of work on improving optimizers using techniques from machine learning by enabling them with learned cost models or learned cardinality estimates or even replacing them entirely with learned models.

An alternative solution is to increase the integration between planning and execution via adaptive query processing. Here, the database system interleaves query optimization and query execution: the database captures information about the currently executing plan and improves it if the plan performs poorly. Taking this to the extreme, the database can eschew a-priori optimization entirely and learn a good plan from scratch for each query by trial and error. This idea forms the basis of SkinnerDB, an in-memory system that uses reinforcement learning to learn join orders from scratch on each incoming query. While giving up prior join ordering entirely comes with overheads in the common case, it can produce orders of magnitude speedups compared to poor query plans chosen by optimizers.

Parallel to these developments in query optimization is query compilation, a technique for increasing the performance of in-memory database systems. When I/O does not dominate query execution, every instruction executed matters for runtime. The database can exploit this by generating machine code specific to each query similar to just-in-time (JIT) compilation systems. This en-

ables much faster execution than standard interpreted engines as the database can generate tight, optimized loops over fused operators which preserves tuple values in registers and avoids unnecessary materialization.

While adaptive execution has been successfully incorporated into interpreted or vectorized engines, a natural tension exists between adaptive and compilation-based processing. This arises from the potentially hundreds of milliseconds that it takes to compile a single query plan when generating a high level language like C++ or even using standard compilation frameworks like LLVM. This cost is expensive for single queries and only grows larger if the database must recompile multiple plans within each query.

To get around the high compilation latency, execution engines have started to adaptively switch between interpretation to avoid the compilation cost for quick queries. For short running pipelines, it will use interpretation but for longer running pipelines, it switches to compiled execution. However, this leads to increased complexity as now the database must juggle two types of execution engines and the performance gap between interpreted and compilation-based execution is high. More recently, Kersten et al. drew from other JIT compilation systems and created a low latency query compilation framework dubbed Tidy Tuples which allows them to match the latency of interpreted systems on even short running queries while only being slightly slower than LLVM [8]. On the other hand, Menon et al. recently worked around the latency of recompiles by designing adaptive optimizations that fit into a compile-once framework [13]. By introducing indirection that can be permuted at runtime, they are able to compile a query plan in such a way that it can be adapted without recompiling. Specifically, they consider adapting the join order for left-deep joins

under certain schema and column access restrictions.

Motivated by these two lines of work, we investigate how to combine SkinnerDB's per-query adaptive processing with a compilation engine with the goal of reducing the overhead of adaptive optimization as much as possible. We create a compile-once permutable approach for adapting join order over a left-deep plan that has none of the aforementioned restrictions. We additionally build a low latency compilation engine and demonstrate that recompilation-based execution is surprisingly practical despite compiling new code for each join plan explored. We experimentally compare these approaches to existing high-performance analytical databases such as MonetDB [6] and DuckDB [17] on a variety of end-to-end benchmarks. Surprisingly, despite using no data statistics or cardinality estimates, we can achieve competitive performance in optimization settings where join ordering is easy while out performing in more challenging settings. In even harder settings, when query rewrites obfuscate information to existing optimizers, our approach dominates.

This thesis is organized as follows: Chapter 2 provides the necessary background on intra-query learning, compilation-based execution and methods for integrating adaptivity into compilation-based systems. Chapter 3 outlines our compilation framework and how it achieves the low latency required for making recompilation practical. Chapter 4 proposes both permutable and recompilation methods for integrating intra-query learning in a compilation engine. Chapter 5 evaluates the proposed methods and Chapter 6 discusses related work.

CHAPTER 2

BACKGROUND

2.1 Intra-query Learning

Intra-query learning is an extreme form of adaptive query processing that assumes that join order planning by using cardinality estimates or other metrics cannot be done prior to query execution. As such, the database must learn good join orders from scratch for each query. It accomplishes this by dividing up join execution into minute time slices called episodes. During each episode, it selects a join order and executes that for a fixed number of computational steps. In doing so, it produces a small fragment of the join output and gains information on the performance of the selected join order. This data can be used to inform future join order selections.

The SkinnerDB system realizes this approach for learning left-deep query plans on the fly and is outlined in Algorithm 1 [22]. It uses reinforcement learning to balance the trade-off between exploring new join orders and exploiting high performing ones and maintains additional data structures that allow for efficiently resuming progress for a specific join order and sharing progress across join orders.

For selecting join orders for each episode, it relies on the UCT algorithm which models a sequence of decisions as a sequence of multi-armed bandit problems. When applied to join ordering, each decision in the sequence corresponds to the next table to join with the set of currently joined tables. The benefit of this algorithm is that it obtains bounded regret, i.e., the difference

Algorithm 1: Intra-query Learning

```
procedure SKINNERJOIN(q, U, S, b, R)
    finished ← false
    t ← 0
    while ¬finished do
        o ← UCTCHOICE(U, t)
        s_prior ← RESTORESTATE(S, o)
        finished, s ← CONTINUEJOIN(R, q, o, b, s_prior)
        UCTUPDATE(U, t, o, REWARD(s, s_prior))
        UPDATESTATE(S, o, s)
        t ← t + 1
    end while
end procedure
```

between the execution time of the best left-deep join order and the cumulative execution time across the history of the join order selections is bounded. This is because the algorithm efficiently manages the trade off between exploration and exploitation. That is, it balances gathering information about a promising but unknown join order with exploiting the highest performing join order it has previously encountered.

To track progress, the system maintains a join order tree where each edge is annotated with a table. Each node contains the last fully completed tuple for the selected partial join order from root to that node. During execution, a join order can resume starting at the tuple immediately after this last completed tuple. It can additionally skip over rows that have an index below the last fully

completed tuple for that table as all rows that include a tuple below that index have already been added to the output.

Finally, to account for the fact that individual episodes can produce previously outputted tuples, it materializes a set of tuple id vectors representing which tuples in the cross product have been added to the result.

2.2 Compilation-based Execution

There are two paradigms currently for building a performant execution engine for in-memory analytical databases: vectorized and compilation-based execution.

In vectorized execution, the database source code contains kernels that execute a specific function over a primitive type, such as filtering an integer column on an equality condition. Given the query plan, the execution engine then chains together these kernels to execute the query.

In contrast, compilation-based execution generates code specific to each incoming query that computes the output. Akin to JIT compilation systems, this code is then compiled and loaded into the database binary after which it is called to compute the query result. While early versions of this technique involved generating a high level language like C++, newer engines directly generate an intermediate representation (IR) like LLVM. This avoids any unnecessary file I/O and expensive compilation passes that convert the high level language into IR.

While compilation engines offer high performance, code generation itself

is a large bottleneck as time spent compiling code is time not spent executing the query. The standard workaround is to introduce multiple execution backends that execute the generated IR such as an interpreter or a standard compiler. By switching between a backend that is quick but inefficient, such as an interpreter or a low-latency compiler, and a more optimized backend that spends additional time generating efficient code, the database can hide the latency of compilation for quick queries.

2.3 Adaptivity and Compilation

Adaptive query processing is a technique where either the query plan or the implementation of the operators in that query plan are varied at query execution time to increase robustness to optimizer mistakes due to unknown or outdated data statistics. The aforementioned intra-query learning adopted by SkinnerDB is an extreme case where a-priori optimization is completely removed. More moderate versions of this technique include switching between a fixed set of plans at runtime or re-optimizing when cardinality estimates differ.

When integrating these techniques into a compilation-based engine, since all query processing takes place in a JIT environment, the execution engine must either compile all plans that the runtime will need before execution begins or somehow reuse prior generated code to avoid recompiling.

Compiling multiple plans means the database is restricted to a small number of possible plans as beyond a certain point, the overhead of compiling the extra plans removes any benefit gained from adaptivity. Nevertheless, this is still viable in scenarios such as select operator implementation where the database

can choose between executing each filtering predicate using a branch or a vectorized select instruction [15]. For larger search space problems, the database must explicitly address the cost of compilation as every new plan tried incurs the compilation overhead.

Permutable compiled queries is a technique introduced by NoisePage that enables reuse of prior generated code by framing the adaptive optimization in such a way that recompilation is unnecessary [13]. This is accomplished by generating code once that uses indirection based on some global state. For example, filter re-ordering can be written in this fashion using a function pointer array where each function evaluates a filter and the order of the array corresponds to the filter order executed by the engine. By permuting this array, the database can execute different filter orders without recompiling. Adaptive optimizations that vary the implementation of a fixed operator can also be expressed in this paradigm. For aggregations, the database can trigger a fast path optimization by checking whether a key is present in a dynamically set hot array. Finally, restricted versions of join ordering can be expressed under this model. NoisePage includes a method of reordering left-deep join plans for a star schema where each table is joined to a single center table. They additionally add the restriction that the corresponding join predicate only references the center table columns. The database compiles a key-check function for each joined table and maintains an array of hash tables and function pointers. As each join can only add or remove tuples from the center table, the database can change the join order similar to the filter case by simply changing the order in which the hash table table look ups and key-check functions are invoked.

CHAPTER 3

COMPILATION FRAMEWORK OVERVIEW

Before describing how the Skinner join algorithm can be integrated in a compilation engine, we first give an overview of our SQL compilation framework largely based on the Tidy Tuples framework [8] and depicted in Figure 3.1. At a high level, a physical query plan passes through the translator layer which generates IR that corresponds to that specific query. This IR then undergoes optimization passes before being passed to an execution backend that translates the IR into executable assembly. This backend for our system is either a lower latency compiler which we call ASM or LLVM. Finally, this executable assembly is invoked to generate the query result. We now go into specifics for each of these components.

3.1 Translator Layer

Each operator in the physical query plan gets an associated translator object which generates IR that computes the result of the plan sub tree rooted at that node. This translator objects keeps track of which IR values map to its output schema. Our code uses a mix of both the produce, consume model [14] and the callbacks model [21] for managing inter-operator interactions. Specifically, we mostly use the produce, consume model to trigger an operator to generate a loop over its result and push tuples to the parent operator but for ad-hoc interactions such as triggering a scan over an individual column, we use the callback model. We divide up the plan into pipelines that correspond to materialization points between operators and generate a function per pipeline. Any data structures such as vectors or hash tables used across pipelines are managed by an ex-

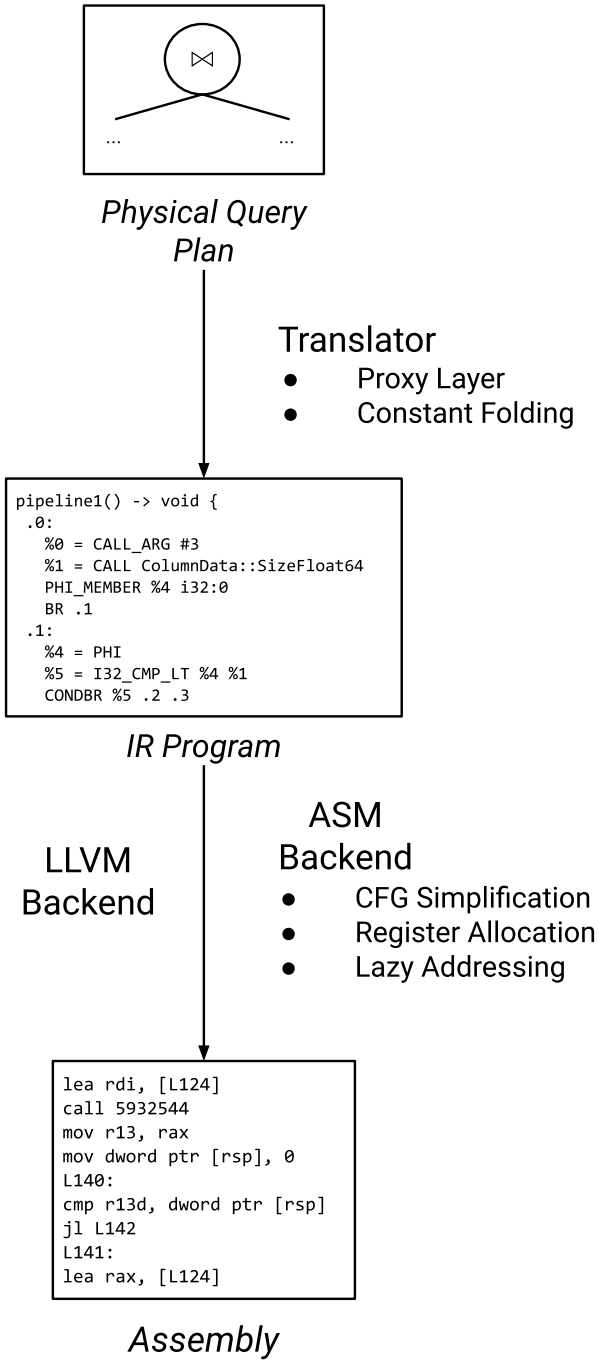


Figure 3.1: Compilation Framework Overview

```
proxy::Bool cond(program, true);
proxy::Int32 x(program, 10);
proxy::Printer printer(program);

proxy::If(program, cond, [&]() {
  printer.Print(x + 10);
}, [&]() {
  // Never invoked
  printer.Print(x + 5);
});
```

Proxy Layer

```
func() -> void {
 .0:
   %0 = CALL_ARG i32:20
   %1 = CALL PrintI32
   RET
}
```

IR Program

Figure 3.2: Proxy If Statement Example

ternal state manager which allows for sharing not only across functions but also across IR builders by injecting them using constant pointers. A pipeline manager keeps track of pipeline dependencies so that during execution, the pipeline executor that invokes each pipeline function knows in which order to call them.

Each translator maintains a reference to an IR builder and generates IR via either directly calling the builder or a generative programming framework similar to the proxy layer used in Tidy Tuples, originally introduced in the LMS framework [18] and used in LB2 [21]. The key idea is to create wrapper classes such as a `proxy::Int32` that represents a 32-bit integer in the IR and stores a reference to the IR builder. Given a `proxy::Int32 x` and `proxy::Int32 y`, writing `x + y` directs the builder to generate an add instruction in the IR. In addition to the standard integer, floating point and pointer types, we additionally provide wrappers for constructing aggregate data types like structs and vectors. Functions and data structures implemented in the database source code can be accessed in the IR by similarly creating a proxy wrapper class where each member function of the wrapper class generates an absolute call to the actual function pointer inside the database binary.

For control flow, we implement functional wrappers for if statements and for loops that generate basic blocks and manage loop variables inside of the IR builder. We perform high-level dead code elimination during code generation by lazily generating constant values in the IR and avoiding code generation for the corresponding branch if the condition is a constant. For example, in Figure 3.2, the proxy layer code is C++ where `program` is an IR builder. During the code generation process, as the constants are lazily translated, creating `cond` and `x` do not result in any generated IR instructions. The implementation of `proxy::If` recognizes that the condition is a constant and only invokes the lambda for the then branch without creating any new basic blocks. The IR layer performs constant folding for the add instruction as we describe below which results in 20 being passed as a constant to the `PrintI32` function.

3.2 IR Layer

For our intermediate representation, we avoid using the sea of nodes representation used by LLVM and instead use a flat vector of 64-bit instructions as outlined by the Tidy Tuples framework and the LuaJIT compiler [12]. Using this instruction representation, basic blocks are represented as a list of contiguous segments of the instruction array and instructions can be referenced by their offset in this array. Each instruction contains an opcode such as `I32_ADD` and encodes all the information needed for that instruction like the set of argument values, the set of basic block labels and output type. For instructions that require more information than can fit in a single 64-bit encoding such as a get element pointer (`GEP`) instruction which is used to index over arrays and index into structs, auxiliary instructions are inserted immediately before the main in-

struction which are ignored when traversing over the IR. Constant values are maintained in a separate instruction array and the value encoding within each instruction marks whether or not the value is a constant. This enables constant folding during IR generation and largely eliminates the need for dead code elimination in the IR.

Similar to LLVM, the IR is in static single assignment (SSA) form where each value is defined only once and phi instructions are used to coalesce multiple values across control flow edges in loops and ternary statements. However, unlike LLVM and as recommended by Tidy Tuples and LuaJIT, the IR builder does not support inserting instructions arbitrarily and is append/update only. Instructions are removed simply by shrinking the corresponding segment in a basic block to exclude them and are updated by overwriting the encoded instruction at that array index. For example, when generating a phi instruction at a branch, each basic block generates a `PHI_MEMBER` instruction which maintains a reference to the corresponding `PHI` instruction and a reference to the input value. This reference is created by first generating the `PHI_MEMBER` with a null `PHI` reference and then when the `PHI` instruction is created, overwriting the previously generated instruction with a reference to the newly created instruction.

The IR builder applies optimizations such as constant folding immediately when building IR as it can identify constant or non-constant values. Before passing to one of the backends, it performs control flow graph (CFG) simplification. This entails removing unused basic blocks, merging chains of basic blocks into a single basic block and removal of redundant phi nodes.

3.3 Backend

As mentioned above, we use either a custom compiler or LLVM to convert the IR into executable assembly.

For LLVM, the conversion is straight forward as each IR instruction corresponds directly to a sequence of LLVM instructions. Once created, we apply instruction combining, global value numbering, CFG simplification and dead code elimination within LLVM before handing it to the ORC API to generate executable x86-64.

For our ASM backend, the goal is to produce much lower latency than LLVM at the cost of producing slightly less efficient code. We utilize the asmjit library as an assembler [9]. We perform no complex tiling of IR instructions and instead map each IR instruction one-to-one to a sequence of x86-64 instructions. Because each value is marked as a constant or not, we can take advantage of constant operands in x86 instructions. For example, the IR instruction `I32_ADD %1, i32 1` will correspond to `add eax, 1` if value `%1` is mapped to register `eax`.

We additionally implement the lazy address calculation optimization mentioned in the Tidy Tuples paper. Since x86 supports complex addressing modes such as `[base + offset]` and `[base + index * scale + offset]`, we do not compute the result of a `GEP` instruction unless required as a function argument or a value to be stored in memory. Instead, for loads and stores, we reference the arguments of the `GEP` to retrieve the base, offset and index/scale parameters. We restrict `GEP` instructions at the IR level to contain at most one non-constant operand to match exactly to these addressing modes.

Each non-constant IR value is mapped to either a physical register or a stack slot using the linear scan register algorithm [16]. For comparisons that are used in conditional branches, we attempt to map that to the flag register. The lazy address calculation is supported by assigning an additional register to any store instruction. Liveness is computed using the standard work list algorithm which is reduced into live intervals over the original instruction order.

3.4 Pipeline Adaptivity

For certain pipelines which simply iterate over its input such as filtering pipelines or scan pipelines, we mark those as split pipelines and have the pipeline function accept the range of which input values to process. This allows for switching between the ASM and LLVM backends to enable the benefits of the low latency ASM backend on short running pipelines and the speed of LLVM on long running pipelines. This technique was previously used to switch between interpreted and compiled execution in [10]. The pipeline executor initially executes the pipeline with the ASM backend and measures the runtime. It then estimates the remaining total pipeline runtime with either the ASM backend or LLVM backend and switches accordingly. Non-split pipelines do not switch backends and either use the ASM or LLVM backend.

CHAPTER 4

COMPILING ADAPTIVE JOINS

We now describe how we incorporate the plan level adaptivity from SkinnerDB into this compilation framework in two ways. First, we build a permutable or compile-once approach that uses state-based indirection to support changing the join plan without compiling additional plans. We also build a recompilation variant that takes advantage of the low latency ASM backend by simply generating assembly code for each new plan that it explores.

4.1 Overview

Input Pipelines

Similar to how a hash join is computed in a compilation engine, the Skinner join translator will start by generating pipelines that materialize all its inputs and build hash tables on the join columns. Note that while a standard hash join plan can avoid materializing one of the tables to use as the base table in the join output pipeline, this is not feasible when the join order can vary at run time. While not necessary for our system, for scans without any filters that are inputs to the Skinner join operator, we can make use of pre-built secondary indexes on each join column and avoid re-materializing each table by just doing random access on the underlying column data.

UCT Implementation

As previously mentioned, we select join orders using the UCT algorithm. As the code for this is common across all queries, we can avoid generating this code for each query by making use of the proxy layer mentioned in Section 3.1. We implement this directly in the database source code under a join executor function and then in each IR builder, declare it as an external function, passing the join executor function pointer.

Within the join executor, as it is in database source code, we can directly use C++ classes to manage data structures that are agnostic to how the join is computed such as the UCT tree, the progress tracking tree as well as the offset array.

High-level Join Implementation

Within the join output pipeline, the Skinner join translator again makes use of the proxy layer to declare data structures that must be accessed within the generated code for the join. This involves the set of tuple id vectors used to deduplicate the output which we maintain as an adaptive radix tree [11]. It initializes these data structures and then invokes the aforementioned join executor.

The join executor accepts as input metadata about the join such as the join graph, the number of tables and the number of predicates, etc. It additionally accepts pointers to each of the required data structures and information specific to each variant that allows it to trigger execution of the chosen join order during each episode.

4.2 Permutable Execution

Motivation

The aforementioned permutable join method for left-deep joins by Menon et. al, makes several assumptions that no longer apply in the more general scenario considered by the Skinner join algorithm. We start by describing these assumptions and how they are invalidated before describing how we overcome them.

- Their algorithm assumes that the center table is fixed and so all joins are between each table and the center table. Additionally, they assume that all column references are restricted to the base table. This means that each join only marks whether or not the tuple from the base table should be removed or kept. When operating on an arbitrary left deep query plan, each join can produce multiple tuples and the base table can change. This necessitates a control flow mechanism beyond simply looping over the base table and filtering it based on the order of join table functions in an array.
- Join predicates can only ever be executed in the non-center table functions while for an arbitrary join plan, the predicates can potentially be executed after any of the tables referenced by the predicate are joined.
- Since predicates are not restricted to reference solely the base table, each function needs to be able to reference any of the previously joined tables' columns.
- We need to be able to suspend a specific join after a certain number of

computation steps to maintain the regret bounds guaranteed by the Skinner join algorithm while in the PCQ join scenario, the individual table functions do not require specific profiling code.

- Since each join with a table is only a function of the center table's columns and since the center table is fixed, there is no need for resuming execution of a specific join order at a specific tuple id vector as each function is stateless. In the Skinner join setting, since the join order completely varies, each join can pause at any point and we must be able to resume execution where it left off.

Nevertheless, we use this as a starting point as the core idea of a compile-once approach is promising and address each of these issues below.

Control Flow

We compile a single function for each table in the join and use a callback style mechanism to organize arbitrary control flow between the tables. We create a function pointer array containing one entry for each table in the join. The ith entry corresponds to the callback for the ith table of the join. Within this function, we generate a loop over the table and invoke the callback for every valid tuple of the current join order prefix. This effectively uses the call stack to represent the current state of the join. For the final table, invoking the callback represents a valid tuple in the output. As such we maintain a global array of tuple ids that contains the current tuple id for each table in the join. Before invoking the next function, the ith function will store the current tuple id in the ith position of the array. The final table will invoke a function that checks if

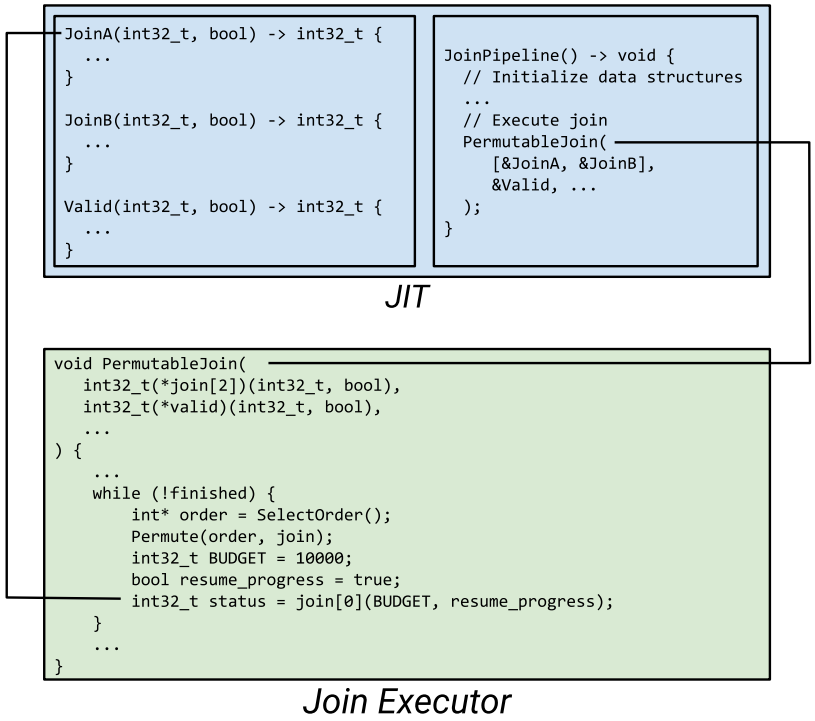


Figure 4.1: Permutable Execution Overview

the current tuple id vector is present within the output set and if not, inserts it, materializes the full join tuple by loading the appropriate columns and calls consume on the parent operator. In the join executor, it can set the join order by setting the next function in this function pointer array.

Predicate Invocation

As any join predicate should be checked immediately after all the tables referenced by the predicate have been joined, under the above control flow mechanism, any join predicate can be executed in any referenced tables' functions. We thus compile it in every referenced table's function but for each instance, guard

it under a flag stored in a global array of booleans. When passing this to the join executor, we pass the pointer to the boolean array but also a set of tables for each predicate as well as for each predicate/table pair, what index in the array corresponds to the flag that that predicate is guarded under in the corresponding table's function. When setting a join order to execute, the join executor iterates over each table and enables every predicate associated with that table if all other tables referenced by the predicate have been joined. Any other flag is set to false skipping execution of that predicate in the corresponding table.

Column Access

Since each table can now refer to any column from a previously joined table, we create a global struct that contains each table's column. At the start of each function, we unpack all columns of the struct and marking each child operator's IR value map to read from the unpacked struct. When loading the next tuple from the current table, we update the IR value map for each child operator to contain the unpacked values and then before invoking the callback, store the current table's columns in the struct. This process can be optimized by noting that each function only requires the table columns that appear in a join predicate.

Budgeted Execution

As the reward function used for selecting join orders is based on how much progress is made executing a specific join order, it is vital that we be able to suspend join execution after a fixed number of computational steps. To track steps, we have each tuple considered by each table's function to count as one compu-

tational step which matches the original SkinnerDB paper. For each episode, we refer to the total number of computational steps as the budget. To keep track of how the budget changes throughout join execution, we make use of the function arguments and return value. Each function accepts as an argument the remaining budget for the current episode. For each tuple it loops over, it decrements this value. Each function returns how much budget is remaining after performing its join and so when a function runs out of budget, it returns zero. When a function invokes a callback, it sets the budget equal to the returned value to account for any computational steps spent joining tuples to the current tuple of the join prefix. The final function for outputting tuples simply returns the passed in budget.

Progress Resuming

A key aspect of the original algorithm is resuming progress for a specific join order at a specific tuple id for each table. Under the above callback control flow model, this corresponds to reconstructing the full call stack where each function is processing a specific tuple from the corresponding table. We accomplish this by passing an additional boolean function parameter to denote whether or not we are resuming progress. If this is true, the function will start processing at the max of the last fully completed tuple of that table and a tuple id read from a global array which is set by the join executor. If false, it starts processing from last fully completed tuple of that table. This flag is marked as a loop variable for each function's loop and on every subsequent iteration is set to false. It is initialized to the conjunction of the function argument and whether or not we actually resumed progress by selecting the last executed tuple as the next one

to process. On every subsequent iteration, it will be false. This loop variable is passed as the resume progress parameter for each callback invocation.

Note that because of progress sharing across join orders, it is not in general true that the tuple id vector the join resumes progress at will be a valid result tuple and so all join predicates must be checked again. However, no additional code changes are needed for this as the predicates will be checked on the first progress resuming iteration. When one of the checked join predicate is false, the corresponding table function will move to the next iteration where the resume progress parameter is now false. Upon returning, the same will happen in the caller's loop.

Finally, when budget runs out, we need to mark whether the current tuple was fully processed or partially processed. We do this by returning a status: -1 representing that we are out of budget and have fully processed the current tuple id vector or -2 representing that we are out of budget and need to backtrack the tuple id vector by one starting from the current table position.

Pseudo code

We combine each of the above elements and present what the translator generates in C style code in Figure 4.2. The specific query is a basic inner join between two tables on an integer column `SELECT * FROM A INNER JOIN B ON A.x = B.x;`. We generate a function for each table `joinA` and `joinB` that represent joining table `A` and `B` respectively.

The callbacks for each table that represent joining all remaining tables with the currently joined tables are stored in the `next` array with index 0 correspond-

ing to the callback for `joinA` and 1 for the callback of `joinB`.

At the core of `joinA` is a loop over the tuples of table `A`. The loop begins at the max of the last fully completed tuple of table `A` specified in the `last_completed_tuple` array or the specific tuple to resume progress set in `progress`. For each tuple, it stores the index of the currently considered tuple in a global array, fetches any predicate columns and decrements the available budget. Since the predicate involved can execute after either table is joined, the function checks if the appropriate flag is set meaning that all available tables for that predicate have been joined. If so, it checks the predicate and if false, moves to the next tuple in table A. Note that when checking the predicate, it compares the current value of `A.x` value to `predicate_columns.b_x` which represents the currently joined value of `B.x`. When all predicates are valid, the function updates the currently joined value of `A.x` by updating `predicate_columns.a_x` and invokes the callback to join the remaining tables. At any point in time, if budget reaches 0, the function returns and sets the appropriate status denoting whether or not the last tuple was fully evaluated or not.

The `joinB` function is symmetric to the `joinA` function except that it reads from each of the join state variables at index 1. When the last table invokes the callback, it invokes `valid_tuple` which checks if the current fully joined tuple has previously been outputted to the parent operator and if not, outputs it.

To set a specific join order, the join executor needs to set the global variables appropriately before invoking the first table's function. For example, for the join order `A B`, the join executor should set the callback array as `next[0] = joinB` and `next[1] = valid_tuple` denoting that for every tuple of `A`, it is then

joined with `B` and then every tuple of that result is added to the join output. Since the predicate is only available after both `A` and `B` are joined, `flags[0] = false` and `flags[1] = true` should be set to denote that the predicate is invoked when joining `B`. If this is the first time that this join order is being executed, then the executor should invoke `joinA(BUDGET, false)` where `BUDGET` is the budget per episode and `resume_progress` is false as we cannot fast forward. If this is not the first time this join order has been executed, then the executor should set `progress` to be the last considered tuple for this join order and should instead call `joinA(BUDGET, true)`. The executor should finally set `last_completed_tuple` to contain the last fully executed tuple for each table.

4.3 Recompiling Execution Overview

Motivation

Prior work explicitly avoids regenerating IR from the plan level because of the high cost of compilation. However, because of the lower latency backend, our framework is in a regime where code generation and compilation is cheap and so compilation time is not a discounting factor. Even still, for recompilation-based execution to be practical, either it needs to improve compilation time or execution time. For the former, in simple join settings, the recompilation environment can potentially speed up by avoiding duplicate predicate generation. For execution, it avoids the multiple layers of indirection from the callback function pointer array to the predicate flags to support join order switching without recompiling. We outline the specifics of this variant along each of the axes of the

```
// Join State
int32_t (*next[2])(int32_t, bool);
int32_t idx[2];
int32_t last_completed_tuple[2];
int32_t progress[2];
bool flag[2];
int32_t backtrack_idx;
struct {
    int32_t a_x;
    int32_t b_x;
} predicate_columns;
TupleIdVectorSet output;

// Column Data
int32_t* A_x; int32_t A_cardinality;
int32_t* B_x; int32_t B_cardinality;

int32_t joinA(int32_t budget, bool resume_progress) {
    int32_t progress_tuple = resume_progress ? progress[0] : 0;
    int32_t tuple_idx = max(progress_tuple, last_completed_tuple[0] + 1);
    resume_progress = tuple_idx == progress_tuple;
    while (next_tuple < A_cardinality) {
        idx[0] = next_tuple;
        int32_t a_x = A_x[tuple_idx];
        budget--;
        if (flag[0] && a_x != predicate_columns.b_x) {
            if (budget == 0) {
                return -1;
            }
            next_tuple++;
            resume_progress = false;
            continue;
        }

        if (budget == 0) {
            backtrack_idx = 0;
            return -2;
        }

        predicate_columns.a_x = a_x;
        int32_t next_budget = next[0](budget, resume_progress);
        if (next_budget < 0) {
            return next_budget;
        }
        budget = next_budget;
        next_tuple++;
        resume_progress = false;
        continue;
    }
    return budget;
}

int32_t joinB(int32_t budget, bool resume_progress) {
    // symmetric to the above
}

int32_t valid_tuple(int32_t budget, bool) {
    if (output.insert(idx)) {
        // output tuple (A_x[idx[0]], B_x[idx[1]])
    }
    return budget;
}
```

Figure 4.2: Generated code for a permutable binary join:
SELECT * **FROM** A **INNER JOIN** B **ON** A.x = B.x;

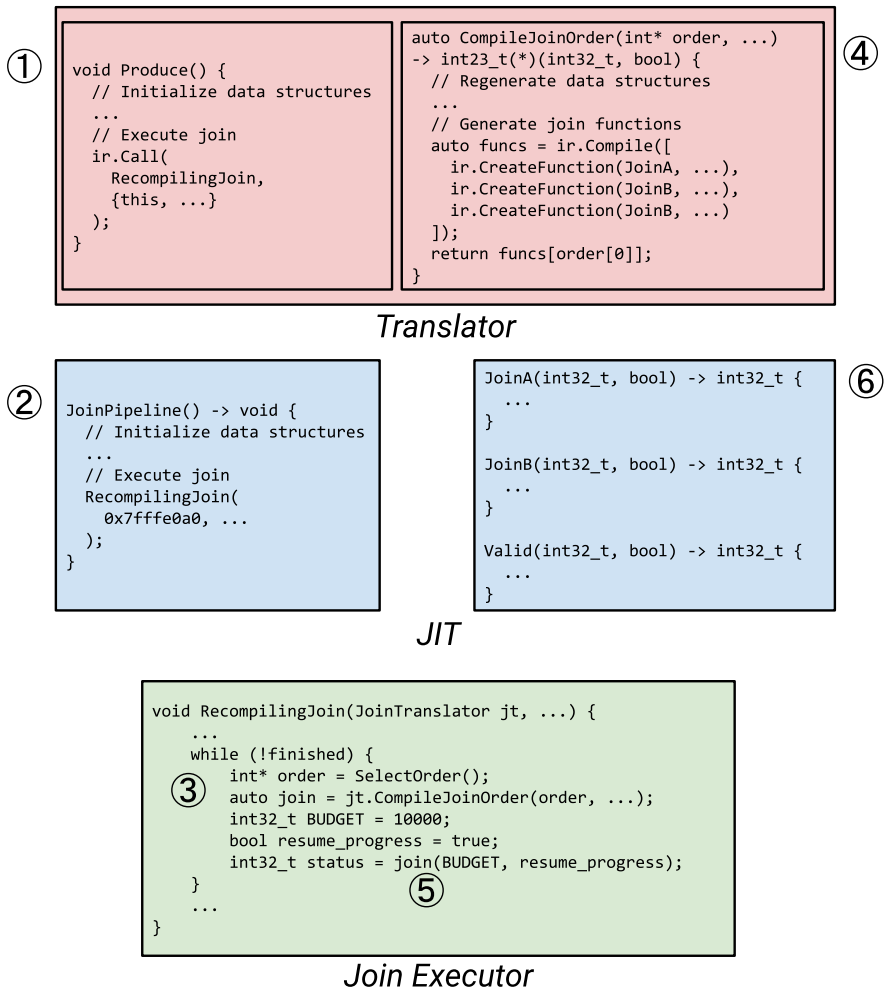


Figure 4.3: Recompiling Execution

permutable variant.

Control Flow

As the join order is known at join order compile time, we no longer need a function pointer array and indirect calls between each of the compiled table functions. Instead, each table function can make use of a direct call to the function of the next table in the join order. While it is possible to inline all the functions,

we choose to preserve the callback structure to make the progress resuming and budgeted execution logic easy to implement.

Predicate Invocation

Since the join order is known at join order compile time, we know in which function join predicates should be evaluated. Thus, we do not need to generate code for each predicate multiple times and we do not need to guard them under a boolean flag.

Column Access

Since we preserve the callback control, we still need the global struct of predicate columns to allow each function to reference previously joined table columns. However, because the join order is known at compile time, we do not need to eagerly unpack the whole struct. Instead, we can lazily unpack the struct as needed during predicate evaluation which leads to only the required columns being loaded for each function. Additionally, we save on memory loads if within a function, a prior join predicate evaluated to false.

Overview

Given this, the recompilation variant has the potential to generate more efficient code which can offset the overhead of compiling a separate set of functions for each new join order that it tries. We now describe how the system can support code regeneration with minimal changes inside of a standard compile-once then

execute framework.

We preserve the translator objects throughout the lifetime of the query. This enables us to reference them from within the join executor to trigger compilation of a new set of join order functions by embedding a constant pointer containing the join translator object pointer and passing that as an argument to the join executor. The join translator presents a `CompileJoinOrder` interface that can be used to trigger compilation of a specific join order. It accepts the join order itself but also any global objects that must be shared across all join order functions as described in the next section. It returns a pointer to the root function of the join order that is used to execute the join order for a specific budget. The translator object maintains a cache of previously generated programs to avoid recompiling the same join order multiple times. For a join order not in the cache, it creates a new IR builder, regenerates each of the global data structures using the below mechanism, generates each of the join functions and lowers it to executable code via a backend.

State Sharing

As we compile separate join order with separate IR builders, we need a mechanism to share query global values across all JIT programs. The tuple id vector set used to deduplicate the output must be the same across all join orders. Similarly, the materialized input tables and hash index objects should be reused across all functions. To accomplish this, we make use of constant pointers in the IR level.

Note that the top level IR builder used for the join pipeline will initialize

the necessary data structures and invoke the join executor and will thus outlive any individual functions used in join evaluation. This means it can manage the lifetime of all values that are required to be shared across all subsequently generated functions. To share references, it passes pointers to each of the aforementioned objects as arguments to the join executor. The join executor then passes these to the translator when triggering the compilation of a new join orders. As we are now in database source code, we can embed these pointers as constant pointers in the newly generated IR by simply augmenting the proxy wrapper classes to use a given pointer instead of allocating a new data structure.

CHAPTER 5

EVALUATION

5.1 Setup

All experiments were conducted on a machine with an AMD Ryzen 7 5800x CPU and 64 GB of RAM on Ubuntu 20.04 with kernel version 5.17.5. As we focus on the compilation aspects of the system and since the original SkinnerDB paper does not support parallelism, we limit all experiments to use a single thread and leave that to future work.

We compare against DuckDB and MonetDB, which feature high performance, vectorized execution engines. For DuckDB, we use version 0.3.4. For MonetDB, we observe performance degradations version to version so we compare against both versions 11.41.21 and 11.43.13. Our system makes use of primary-foreign key indexes for join evaluation. We evaluate DuckDB both with and without indexes. MonetDB automatically creates indexes and ignores any index creation statement. MonetDB uses dictionary encoding (DC) for string columns while DuckDB does not so we report numbers for our system with dictionary encoding enabled and disabled.

5.2 Query Latency

We evaluate end-to-end query latency of each variant including compile and execution time across a variety of benchmarks which stress join evaluation in varying intensities. The variants tested are the recompiling and permutable

Database	Runtime(s)
ASM Permute	51.6 ± 0.2
ASM Recompile	53.9 ± 0.1
LLVM Permute	59.8 ± 0.2
LLVM Recompile	182.8 ± 1.4
DuckDB	110.3 ± 0.1
DuckDB + Indexes	266.3 ± 0.1
ASM Permute + DC	40.7 ± 0.2
ASM Recompile + DC	43.5 ± 0.2
LLVM Permute + DC	49.5 ± 0.2
LLVM Recompile + DC	176.3 ± 2.3
MonetDB (11.41.21)	55.7 ± 0.1
MonetDB (11.43.13)	76.6 ± 0.2

Table 5.1: Total Runtime - JOB

variant each using the ASM backend or LLVM backend.

5.2.1 Join Order Benchmark

The Join Order Benchmark (JOB) is a series of 113 queries over the Internet Movie Database schema which describes movies, actors, etc [11]. Each query consists of a join of optionally filtered tables directly into a flat aggregation and so primarily stresses join planning and execution. We report total runtime across all queries for each variant as well as the baselines in Table 5.1.

DuckDB

For the comparison with DuckDB, we first note that adding indexes to DuckDB surprisingly hurt performance significantly. This is likely because DuckDB's optimizer is relatively new and does not properly do cardinality estimation which means on certain queries, it chooses an incorrect join plan resulting in a severe

slowdown. This occurs on queries 12b, 13a-d, 26c, 30a-c, 31a-c, 33a-c which have a latency increase between 10.1x and 72.5x when indexes are added. This corresponded to a total of 155 seconds of added latency which matches the total latency gap observed. Therefore, we focus on the version without indexes below.

For the permutable variants, the ASM backend achieved a 2.16x speedup while the LLVM backend achieved a 1.86x speedup compared to the non-indexed DuckDB variant. For the recompilation variants, the LLVM backend resulted in much larger execution time which is as expected: the high cost of compiles combined with compiling new code for each plan is prohibitively expensive. However, when combined with the lower latency ASM backend, the recompilation variant achieves a 2.04x speedup.

To identify the source of the overall speedup, we zoom into a per-query chart in Figure 5.1. For all but one query on the LLVM permutable variant, all queries that took longer than 1 second resulted in a speedup. For the remainder of the queries where we slowed down, the runtime of the query in DuckDB was less than 1 second which means that it had negligible impact on the total runtime when accounting for the large speedups on longer running queries.

MonetDB

For MonetDB, note that the later version has significant performance reductions resulting largely from two queries: 20a and 26c which incurred latency increases of 7.5x and 15.45x from version 11.41.21 to 11.43.13. This is an example of how tweaking static optimizers to improve performance somewhere can result in

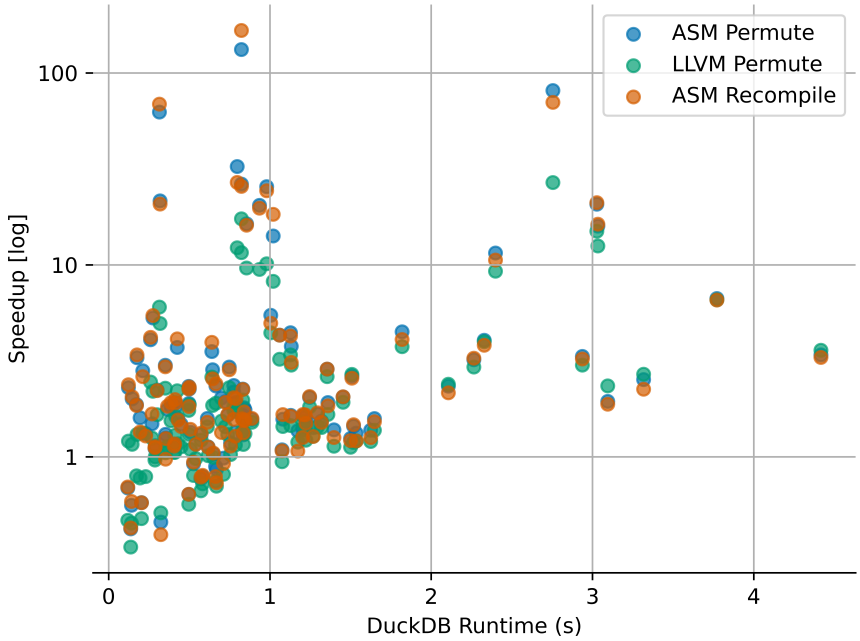


Figure 5.1: Per-query speedups of each JOB query (log scale) vs. DuckDB runtime

drastic performance degradations elsewhere. Nevertheless, we compare against the faster variant (version 11.41.21) below.

For the permutable variants, the ASM backend achieved a 1.36x speedup while the LLVM backend achieved a 1.12x speedup compared to MonetDB. For the recompilation variants, the LLVM backend again resulted in much larger execution time but when combined with the lower latency ASM backend, the recompilation variant achieves a 1.28x speedup.

We plot the per-query speedups versus MonetDB query latency in Figure 5.2. Note that several queries resulted in a slow down which is likely due to MonetDB's much more mature optimizer compared to DuckDB. MonetDB selecting a high performing query plan is a scenario that our approach cannot match

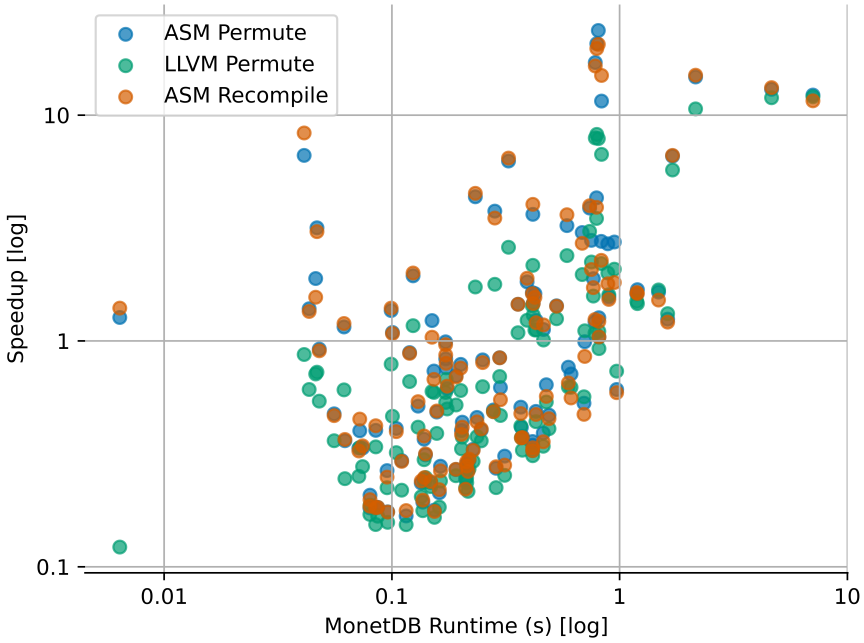


Figure 5.2: Per-query speedups of each JOB query (log scale) vs. MonetDB 11.41.21 runtime (log scale)

as we have to learn a fresh query plan from scratch on each invocation of the query. As they do not have this overhead, they can spend that time executing the join. However, this is offset by the drastic speedups achieved on long running queries where MonetDB selects an incorrect query plan. For example, on MonetDB's longest running queries, 6f and 6d, we achieve a 10.65 and 11.79 speedup respectively.

Robustness

To show how a classical mature optimizer is reliant on unsound heuristics, we generate a variant of JOB where we insert redundant predicates. We select four tables in the query and either add the predicate `table.join_col >= 0` five

Database	Strategy		
	Original (s)	Smallest (s)	Largest (s)
ASM Permute + DC	40.6 ± 0.1	51.1 ± 0.2	240.8 ± 0.4
MonetDB (11.43.13)	78.4 ± 0.1	69.3 ± 0.1	2222.9 ± 10.8

Table 5.2: Total Runtime - JOB with additional predicates

times if that table has no unary predicates applied or if it does have unary predicates, repeat the same unary predicate five times. The addition of these predicates has no impact on the query output or optimal query plan. We choose tables based on the largest four tables after unary predicates are applied or the smallest four tables after unary predicates are applied. Note that we do not apply any form of common sub expression elimination. The results are displayed in Table 5.2.

We select the permutable variant with the ASM backend as this had the highest performance on the unmodified benchmark. When adding additional predicates to queries on our system, we expect runtime to increase because of the additional time needed to scan those tables to check those predicates as well as the additional time to build up hash tables on all join columns to enable join order learning. However, as we learn join orders on the fly, we expect that the average time spent executing the join should remain the same regardless of any added predicates. This is what we observe as when adding predicates to the smallest tables, we incur a slight overhead of 1.25x increase and when adding to the largest tables, we incur a larger 5.92x increase.

In theory, the performance of a classical database system that uses an optimize-then-execute approach should not drastically increase as the predicates do not affect the tables that input into the join and so the optimal join order remains the same. The only additional overhead would be that of scanning the

tables to check the predicates and the cost of hash table construction. However, in practice, this is not what is observed. When adding redundant predicates to the smaller tables, MonetDB's runtime actually improves. A large improvement comes from query 26c where it originally took 14.96 seconds to execute but only takes 2.78 seconds with the predicates added. Examining the plan reveals that MonetDB pushed the filtered tables deeper into the join resulting in a better join order. In contrast, when adding predicates to the largest tables, the opposite occurs. MonetDB runtime increases by 28.35x compared to the original. The slowdown comes from queries 16a-d and 17a-f which incur between a 7x and 36x latency increase for the opposite reason as mentioned previously.

5.2.2 TPC-H, JCC-H

The TPC-H benchmark is a standard analytical query benchmark that features a variety of queries with complex filtering, varying number of joins, grouping and aggregation where data is generated from a uniform distribution. We use queries 1-3, 5-12, 14, 18, and 19 from the benchmark as the remaining queries contained features that we do not support. We evaluate our framework on both scale factor 1 and scale factor 10 to understand performance at different database sizes.

In general, this is not a scenario in which adaptive join processing should be expected to outperform as existing optimizers can easily select a good query plan. This means that we have the overhead of learning a query plan from scratch on each query without the possibility of an optimizer mistake to offset these common case overheads.

SF 1		SF 10	
Database	Runtime(s)	Database	Runtime(s)
ASM Permute	1.80 ± 0.01	ASM Permute	22.5 ± 0.5
ASM Recompile	1.88 ± 0.02	ASM Recompile	21.9 ± 0.1
LLVM Permute	2.32 ± 0.01	LLVM Permute	21.4 ± 0.1
LLVM Recompile	3.87 ± 0.05	LLVM Recompile	23.4 ± 0.3
DuckDB	3.43 ± 0.00	DuckDB	41.5 ± 0.1
DuckDB + Indexes	3.83 ± 0.01	DuckDB + Indexes	51.5 ± 0.1
ASM Permute + DC	1.57 ± 0.01	ASM Permute + DC	19.2 ± 0.1
ASM Recompile + DC	1.62 ± 0.02	ASM Recompile + DC	19.5 ± 0.2
LLVM Permute + DC	2.08 ± 0.01	LLVM Permute + DC	19.3 ± 0.4
LLVM Recompile + DC	3.56 ± 0.07	LLVM Recompile + DC	21.7 ± 0.4
MonetDB (11.41.21)	1.65 ± 0.01	MonetDB (11.41.21)	16.9 ± 0.1
MonetDB (11.43.13)	1.64 ± 0.01	MonetDB (11.43.13)	16.7 ± 0.1

Table 5.3: Total Runtime - TPC-H

The JCC-H benchmark is a variant of the TPC-H benchmark that retains the same query templates but augments the data and query constants to include skew and join crossing correlations [3]. With this addition, our framework is in a position to outperform if existing optimizers make a drastic mistake. Again, we use queries 1-3, 5-12, 14, 18, and 19 from the benchmark and evaluate our framework on both scale factor 1 and scale factor 10 to understand performance at different database sizes.

We start by reporting total runtimes for each benchmark and variant in Tables 5.3 and 5.4. This shows that we achieve competitive performance to existing engines as we achieve comparable performance on TPC-H SF1 to MonetDB and achieving 1.9x faster performance compared to DuckDB. The same trends hold for SF10 except we are now slightly slower than MonetDB. For JCC-H, we achieve comparable performance to DuckDB on both SF1 and SF10. We achieve comparable performance to MonetDB (11.41.21) on SF1 and are 1.4x slower than MonetDB (11.43.13). On SF10, we are 6x faster than MonetDB (11.41.21) and are

SF 1		SF 10	
Database	Runtime(s)	Database	Runtime(s)
ASM Permute	3.22 ± 0.05	ASM Permute	40.9 ± 0.4
ASM Recompile	3.30 ± 0.06	ASM Recompile	41.4 ± 0.4
LLVM Permute	3.62 ± 0.03	LLVM Permute	39.3 ± 0.5
LLVM Recompile	4.85 ± 0.07	LLVM Recompile	40.9 ± 0.6
DuckDB	3.26 ± 0.01	DuckDB	38.1 ± 0.2
DuckDB + Indexes	3.57 ± 0.01	DuckDB + Indexes	42.2 ± 0.1
ASM Permute + DC	2.9 ± 0.04	ASM Permute + DC	36.8 ± 0.4
ASM Recompile + DC	2.97 ± 0.06	ASM Recompile + DC	38.1 ± 0.6
LLVM Permute + DC	3.30 ± 0.02	LLVM Permute + DC	36.7 ± 0.4
LLVM Recompile + DC	4.40 ± 0.09	LLVM Recompile + DC	36.2 ± 0.9
MonetDB (11.41.21)	2.82 ± 0.01	MonetDB (11.41.21)	220.5 ± 13.2
MonetDB (11.43.13)	1.97 ± 0.00	MonetDB (11.43.13)	20.8 ± 0.1

Table 5.4: Total Runtime - JCC-H

1.76x slower than MonetDB (11.43.13). To understand why, we drill down into per-query latency plots in Figures 5.3, 5.4, 5.5 and 5.6.

Broadly, the performance of each of the variants is roughly the same. This is because although all queries but query 1 include a join, the join order space is relatively easy and so the differences between compiling a single program for each join and permuting a single program get washed out. The only notable exception is the LLVM recompilation variant as it once again demonstrates that compiling a new program for each join order tried with an expensive compilation framework does not scale. Similarly, we see that on low data sizes at scale factor 1, the LLVM permutable variant performs slightly worse than the ASM variants as the cost of compilation exceeds the execution cost. This disappears at scale factor 10 as the cost of execution dominates the cost of compilation.

On both JCC-H and TPC-H, on queries 1, 6 and 19, our system outperforms both DuckDB and MonetDB. For query 1 which contains no joins, this is a byproduct of the fact that we use a compilation-based execution engine rather

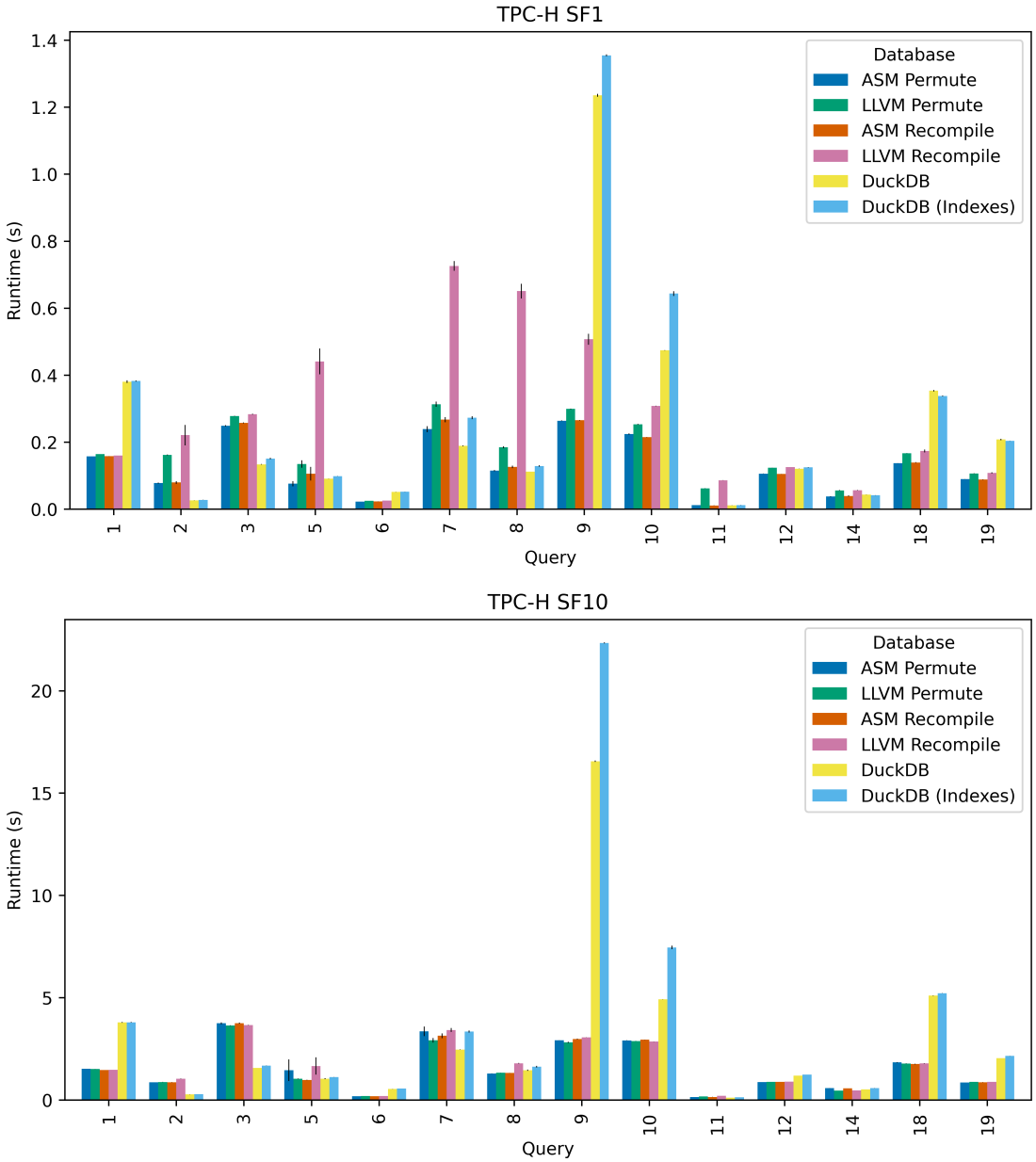


Figure 5.3: Latency of each TPC-H query vs. DuckDB at SF1, SF10

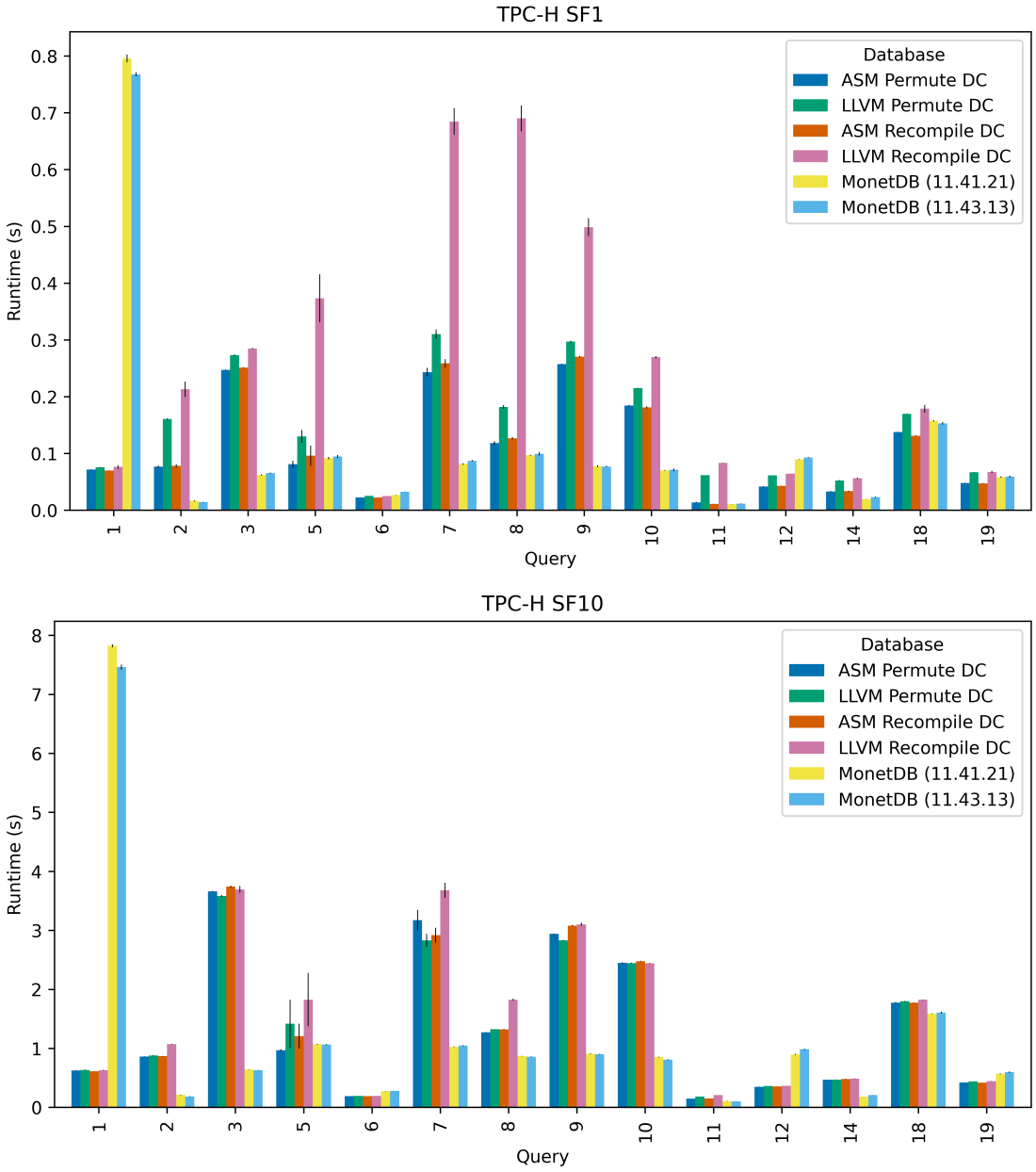


Figure 5.4: Latency of each TPC-H query vs. MonetDB at SF1, SF10

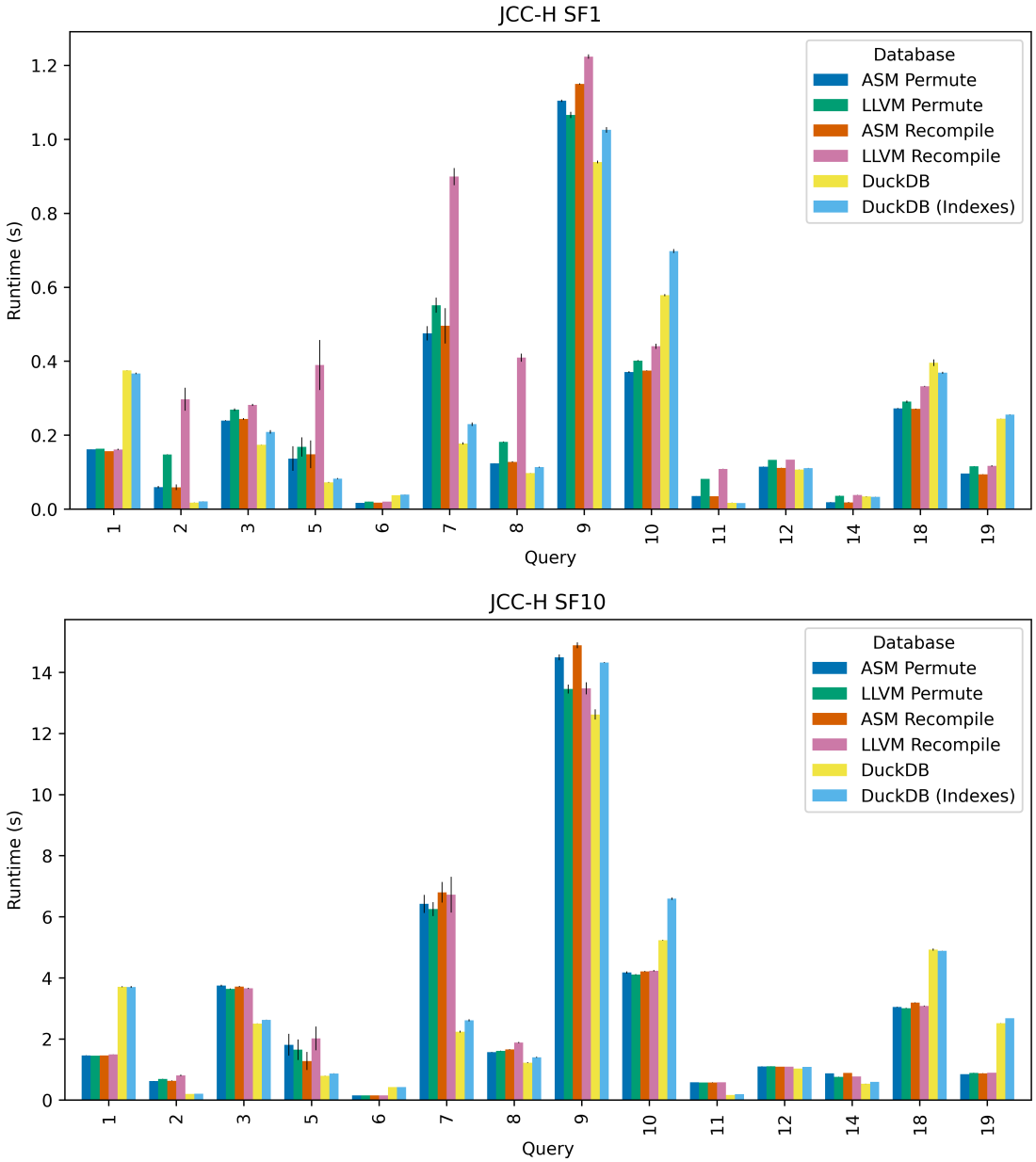


Figure 5.5: Latency of each JCC-H query vs. DuckDB at SF1, SF10

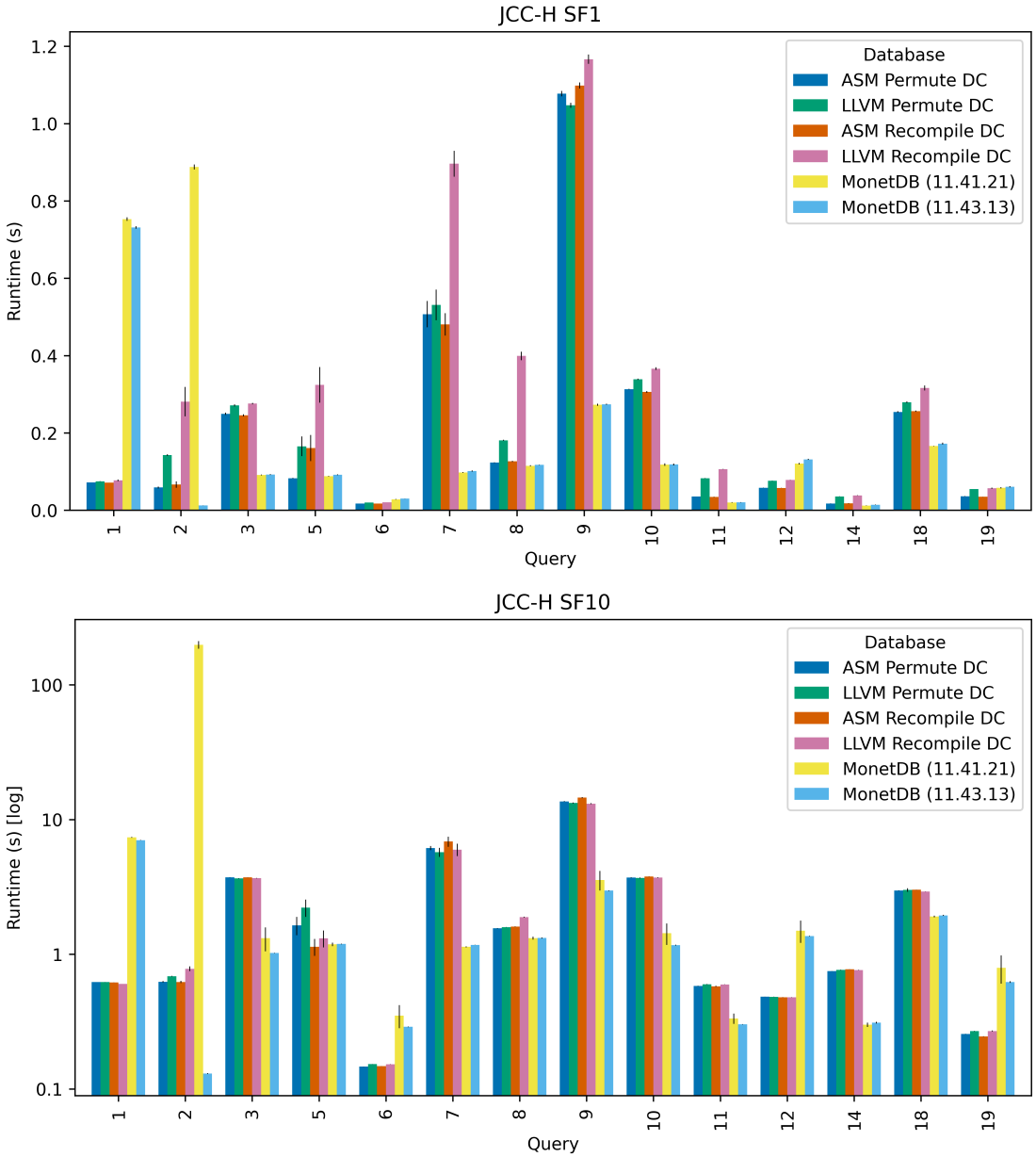


Figure 5.6: Latency of each JCC-H query vs. MonetDB at SF1, SF10 (log scale)

than a vectorized engine. Because we generate tuple-at-a-time code, we avoid materializing any intermediate result while both vectorized engines have some materialization [7]. Q6 is similar to query 1 in that it contains no joins and so we can outperform for the same reason. Q19 features complex filtering conditions which means that the improvement likely comes from our use of branch-based filter execution and choice of predicate order. Prior experiments show that the difference between join order in Q19 which only uses two tables has a relatively small impact on the final query time [4].

The only other notable performance improvement comes from MonetDB (11.41.21) which is 319x slower than the ASM permutable variant on query 2 of JCC-H SF10. Both versions tested flatten the sub-query but have different join orders and sub-plan reuse and is fixed in the later version of MonetDB (11.43.13).

The queries that have a notable slowdown compared to baselines are queries 3, 7, 9 and 10.

For query 3, our system spends 70% of the query in filtering the base tables and building hash tables on each of the join columns. In particular, because we are forced to materialize all inputs fully, we cannot pipeline any of the join inputs and so we end up building more hash tables than the baseline systems. Further, we are limited to building a hash table per column to enable arbitrary join order learning which accounts for even more hash tables compared to the baseline systems. Query 10 also features this issue as 50% of the query is spent filtering and building hash tables while the remainder is spent evaluating the join.

For query 7, 62% of the time is spent in the join phase with the remainder in the prepossessing phase to build hash tables. This adaptivity does not pay off as existing optimizers can find a good join order in both the regular and skewed versions.

Query 9 features a high cardinality join output piped into an aggregation. To enable join order learning, we have to materialize the tuple indexes of outputted tuples to avoid outputting duplicates which is unnecessary in baseline systems. This accounts for 35% of the query execution time while another 35% is spent doing hash table lookups which is an area that we have not spent much time optimizing. Notably, DuckDB on query 9 on TPC-H does much worse than both our system and MonetDB due to an incorrect join order.

5.3 Compilation Latency

To validate that the ASM backend actually compiles faster than the LLVM backend and to compare the execution performance of both, we compare the time spent compiling in each backend and the time spent actually executing the query. We fix the seed used for random exploration so that all implementations follow the same sequence of episodes.

We display the runtime breakdown for JOB in Figure 5.7. We can see that both the recompilation and permutable variants experienced a compilation time increase for the LLVM backend. The ASM backend generates slightly less efficient code as the execution phases are mostly the same but achieves negligible compile times. Even for the recompilation variants which compile a program for each join order tried, the LLVM backend experienced approximately the same

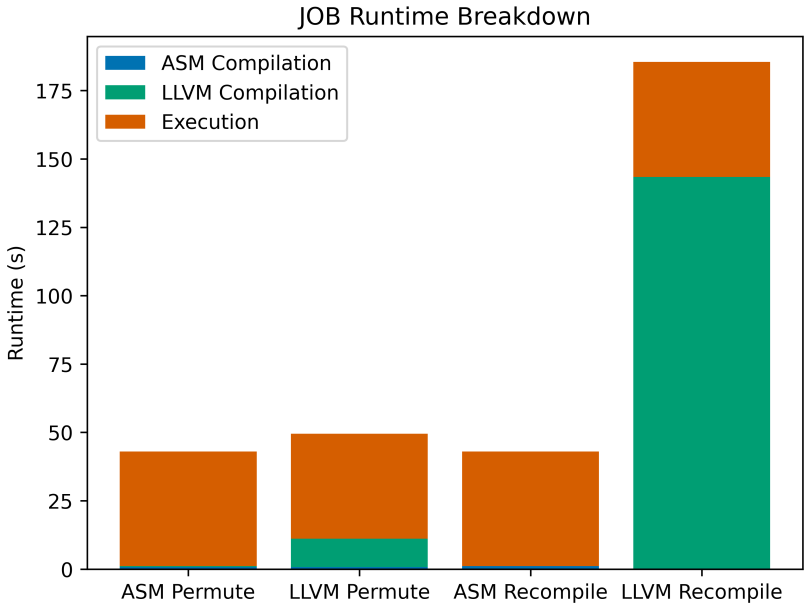
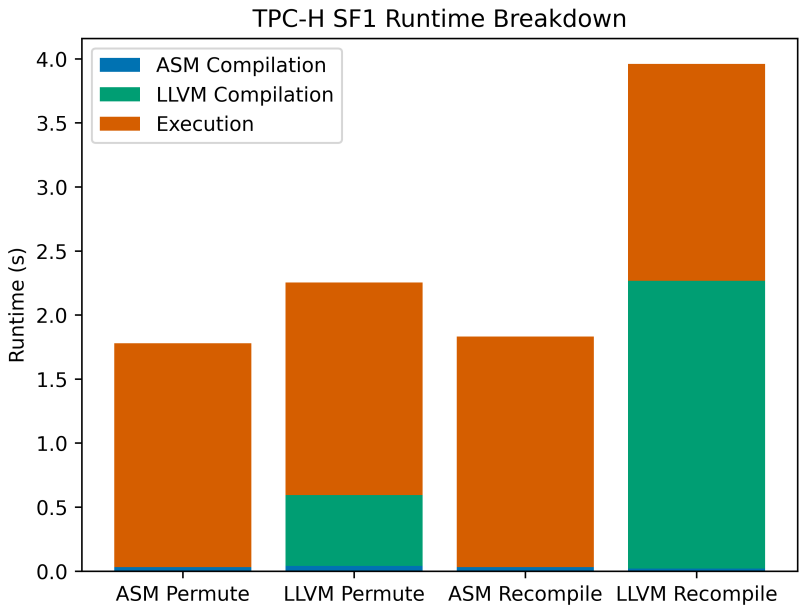


Figure 5.7: JOB Runtime Breakdown

execution time as the recompilation variant but was completely outclassed in compilation time.

We display the runtime breakdown for TPC-H in Figure 5.8 and for JCC-H in Figure 5.9. For scale factor 1, we observe the same trend as for JOB where the ASM backend generates roughly equivalent quality code to the LLVM backend but incurs almost no compilation time. For scale factor 10, the data is at a size where the differences in code quality are noticeable. In particular, the LLVM permutable variant spends a noticeable amount of time compiling but generates faster code leading to a runtime improvement compared to the ASM permutable variant. The LLVM recompilation variant, while roughly around the same latency, still is inhibited by compile time while the ASM variant spends negligible time compiling. We can see that both the recompilation and permutable variants experienced a compilation time increase for the LLVM backend. The ASM backend generates code of comparable quality as the execution



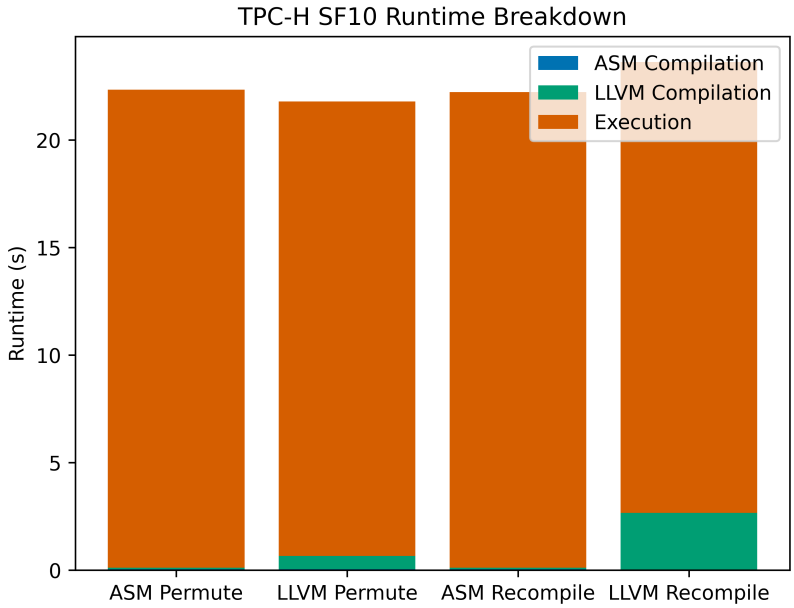


Figure 5.8: TPC-H Runtime Breakdown

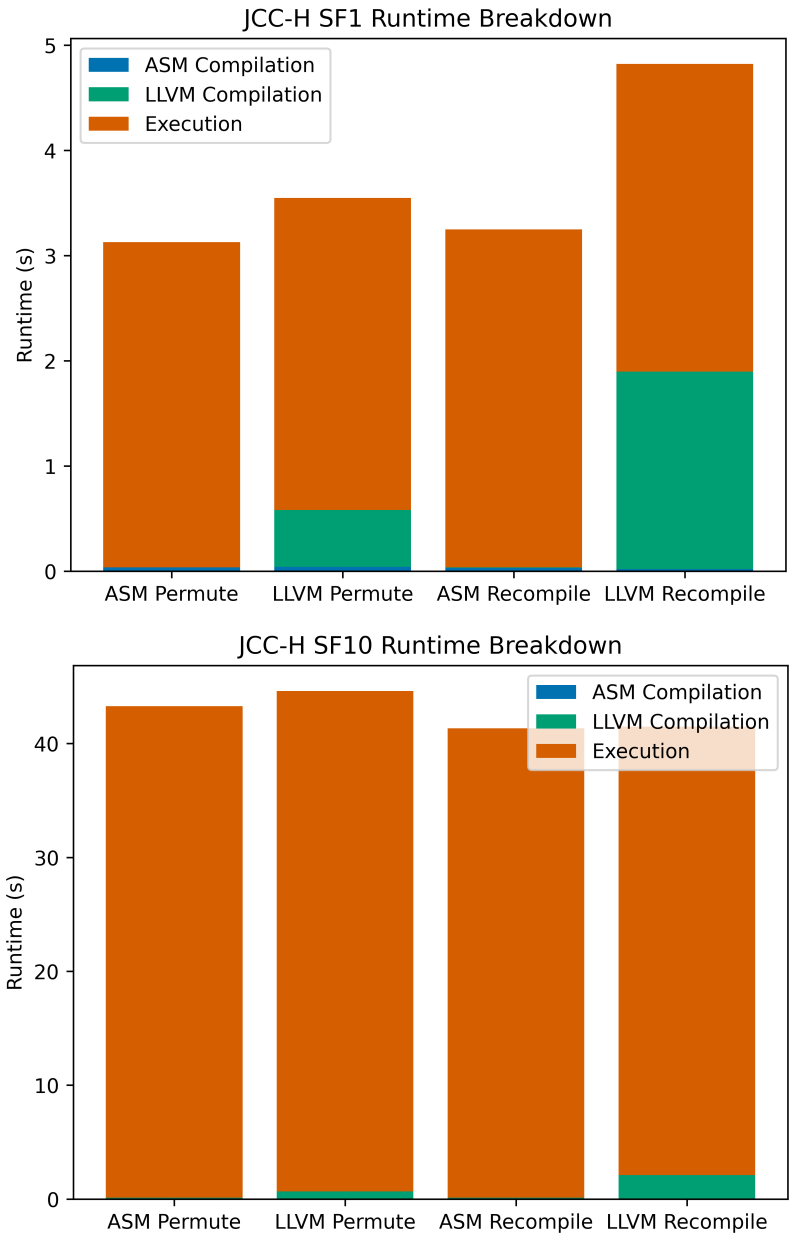


Figure 5.9: JCC-H Runtime Breakdown

phases are mostly the same but achieves negligible compile times. Even for the recompilation variants which compile a program for each join order tried, the LLVM backend experienced slightly faster execution time than the ASM backend but was completely dominated in compilation time.

CHAPTER 6

RELATED WORK

Adaptive Execution

Adaptive execution has long been explored as a method of increasing robustness to optimizer mistakes or handling incorrect or unknown data statistics. Babu et al. surveyed adaptive optimization techniques and divide them into plan-based and routing-based categories [2]. Plan-based systems still use an optimizer to generate the best initial plan but then adapt at runtime if runtime statistics like intermediate result sizes are significantly worse than those predicted. Routing-based systems choose how to process tuples through a series of operators on a per-tuple or per-batch basis. They eschew a traditional optimizer and instead use runtime statistics to measure plan performance. Systems under this paradigm include Eddies introduced by Avnur et. al [1] which re-routes tuples through operators on a per-tuple basis and SkinnerDB [22] which forms the basis of the adaptive join execution we consider.

Other systems in the intersection of plan-based and routing-based are Vectorwise and NoisePage. Vectorwise is a vectorized analytical database that included micro-adaptivity: the database compiles each kernel under different compilers and settings and selects the best performing variant by using a multi-armed bandit algorithm [19]. NoisePage is an analytical database that includes a routing-based join algorithm in restricted settings that is initialized with the result of an optimizer [13].

Adaptivity and Compilation

In the intersection of compilation-based execution and adaptivity, the most relevant work is Permutable Compiled Queries (PCQ) [13]. They propose a method for integrating plan and operator-implementation adaptivity into their database NoisePage without recompiling by introducing global state that is permuted to change the plan. They implement filter re-ordering, fast paths for hot keys in aggregation and reordering left-deep join plans where each table is joined independently to a single center table subject to predicate restrictions. We focus solely on the join case and build a permutable compilation algorithm that allows for searching over arbitrary left-deep join plans without any schema or predicate restrictions.

HyPer includes adaptivity in the execution engine as it switches between compiled and interpreted execution for a fixed query plan [10]. This allows them to avoid the high compilation latency of LLVM for short running queries. More recently, Schmidt et al. extend this to include the above adaptivity techniques from PCQ in the HyPer successor Umbra [20]. Similar to HyPer, Umbra adaptively switches between a fast compiler and LLVM to hide the latency of compilation for short queries. To avoid recompiling queries, they compile an exploration phase into their unoptimized backend at the IR level by introducing special dynamic basic blocks: an alternative block that executes one of many basic blocks, an optional block that optionally executes a basic block and a variant block that arbitrarily reorders basic blocks. When switching to LLVM, they remove adaptivity and compile the best performing implementation. We instead learn throughout the query to maintain regret bounds on the join order selected and utilize standard mechanisms such as indirect calls to enable the indirection

with less complexity in the compilation framework.

Several works explore optimizations outside of joins and are thus complementary to our work. Weld integrates JIT compilation using LLVM with adaptive optimizations [15]. They switch between using branches to evaluate ternary expressions and unconditionally evaluating both cases and then selecting depending on a cost model and selectivity estimates. For parallel aggregations, they also adaptively choose between using a single global hash table and thread-local hash tables that are merged together. Grizzly dynamically adapts predicate order, adapts between thread local and a single global hash table for aggregations, and dynamically chooses between using a static array or a hash table with a speculative optimization on the values range for aggregation [5]. Their framework initially uses a generic variant for an exploration phase and then recompiles new variants by generating C++, reverting back to the original if the assumptions were invalidated.

CHAPTER 7

CONCLUSION

This thesis integrated the per-query adaptive query optimization of SkinnerDB with a customized compilation-based execution engine with the goal of matching databases with high performance execution engines and static optimizers on queries where they make no optimizer mistakes.

The compilation engine features an intermediate representation tailored for fast and efficient optimizations and a machine code backend which can match performance of existing compilation frameworks like LLVM despite spending much less time compiling. We proposed two methods of integrating adaptive query optimization into this engine: a compile-once approach that uses global state and indirection to allow permuting the join order without recompiling and a recompilation approach that compiles a new program for each join order explored during execution.

We evaluated these techniques on three benchmarks and compared against DuckDB and MonetDB, two high performance databases which do not use query compilation but feature vectorized execution engines. The experiments showed that we can match or incur slight slowdown for easy to optimize benchmarks but achieve multiple times faster performance in more difficult optimization settings. We additionally validate that the low latency query compilation framework generates code of a similar quality to that of LLVM while being orders of magnitude faster in compilation time which is critical to enable equivalent performance in the recompilation variant.

To conclude, our results show that even extreme forms of adaptive query

optimization that give up a-priori optimization entirely can be successfully integrated into a high performance database when paired with a tailored execution engine.

BIBLIOGRAPHY

[1] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 261–272, New York, NY, USA, 2000. Association for Computing Machinery.

[2] Shivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 238–249. www.cidrdb.org, 2005.

[3] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. Jcc-h: Adding join crossing correlations with skew to tpc-h. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking for the Analytics Era*, pages 103–119, Cham, 2018. Springer International Publishing.

[4] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. Quantifying tpc-h choke points and their optimizations. *Proc. VLDB Endow.*, 13(8):1206–1220, apr 2020.

[5] Philipp M. Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. Grizzly: Efficient stream processing through adaptive query compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2487–2503, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[7] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, sep 2018.

[8] Timo Kersten, Viktor Leis, and Thomas Neumann. Tidy tuples and flying start: Fast compilation and fast execution of relational queries in umbra. *The VLDB Journal*, 30(5):883–905, sep 2021.

[9] Petr Kobalicek. Asmjit. `https://asmjit.com/`, 2022. Accessed: 2022-05-18.

[10] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 197–208, 2018.

[11] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.

[12] Luajit 2.0 ssa ir. `http://wiki.luajit.org/SSA-IR-2.0`, 2018. Accessed: 2022-05-16.

[13] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proc. VLDB Endow.*, 14(2):101–113, oct 2020.

[14] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, jun 2011.

[15] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. Evaluating end-to-end optimization for data analytics applications in weld. *Proc. VLDB Endow.*, 11(9):1002–1015, may 2018.

[16] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, sep 1999.

[17] Mark Raasveldt and Hannes Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, page 127–136, New York, NY, USA, 2010. Association for Computing Machinery.

[19] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in

vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1231–1242, New York, NY, USA, 2013. Association for Computing Machinery.

[20] Tobias Schmidt. Adaptive query execution: Dynamically rewriting compiled queries. Master's thesis, Technical University of Munich, 2021.

[21] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 307–322, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *ACM Trans. Database Syst.*, 46(3), sep 2021.