Adaptive Code Generation for Data-Intensive Analytics

Wangda Zhang
Columbia University
zwd@cs.columbia.edu

Junyoung Kim
Columbia University
junyoung2@cs.columbia.edu

Kenneth A. Ross
Columbia University
kar@cs.columbia.edu

Eric Sedlar
Oracle Labs
eric.sedlar@oracle.com

Lukas Stadler
Oracle Labs
lukas.stadler@oracle.com

ABSTRACT

Modern database management systems employ sophisticated query optimization techniques that enable the generation of efficient plans for queries over very large data sets. A variety of other applications also process large data sets, but cannot leverage database-style query optimization for their code. We therefore identify an opportunity to enhance an open-source programming language compiler with database-style query optimization. Our system dynamically generates execution plans at query time, and runs those plans on chunks of data at a time. Based on feedback from earlier chunks, alternative plans might be used for later chunks. The compiler extension could be used for a variety of data-intensive applications, allowing all of them to benefit from this class of performance optimizations.

PVLDB Reference Format:
Wangda Zhang, Junyoung Kim, Kenneth A. Ross, Eric Sedlar, and Lukas Stadler. Adaptive Code Generation for Data-Intensive Analytics. PVLDB, 14(6): 929-942, 2021.
doi:10.14778/3447689.3447697

1 INTRODUCTION

The increasing main-memory capacity of contemporary hardware allows query execution in a database management system (DBMS) to occur entirely in RAM. Analytical query workloads that are typically read-only need no disk access after the initial load. In response to this trend, several commercial and research DMBSs have been designed (or re-designed) for memory-resident data [18]. Examples of recent systems include H-Store/VoltDB [29], Hekaton [41], HyPer [32], IBM BLINK [5], DB2 BLU [53], SAP HANA [19], Vectorwise [70], Oracle TimesTen [39], MonetDB [7], HYRISE [20], Peloton [48], HIQUE [37], LegoBase [35] and Quickstep [47]. A variety of advanced query processing and optimization techniques have been developed in these and other systems, several of which we will discuss in detail later in this paper.

Other data-intensive applications have also scaled to the point where they are processing very large RAM-resident data collections. Examples include data visualization systems [61], stream processing systems [11], time-series analysis systems [25] biological sequence processing systems [62] and array processing systems [60]. Many of these applications require DBMS-like functionality, such as scanning, filtering, cross-referencing (joining), and aggregating data. Most of these applications do not use a DBMS as the underlying data storage framework. This choice could be for performance reasons, or because relational tables are not the most natural abstraction for the data being modeled by the application. Nevertheless, the low-level operations (scanning, aggregating etc.) can still potentially benefit from the kinds of optimizations done in state-of-the-art DBMSs.

Additional applications may be written by programmers who do not want the overhead of dealing with an external application. Instead they simply write direct code to store and process arrays of data. For example, a weather analysis application may store data about rainfall measurements. Suppose that the application records the output from a large collection of field sensors that each report rain accumulations each minute, but only when the measurement is nonzero. The data is represented using three arrays: `ID[i]` that represents the identifier of the sensor making the measurement, `time[i]` that represents the time the measurement was taken, and `rain[i]` that represents the actual rain measurement. The `time[i]` values are nondecreasing, reflecting incremental appends of new measurements over time. When there are many sensors spread over a large geographic region, there may be billions of data values stored per day. While there are alternative (e.g., partitioned/sharded) representations of this data, the given representation is actually well-suited if the common query pattern is something like "where and how much has it rained in a given time interval?" Such queries could drive the generation of real-time animations of recent (or historical) rainfall. A query coded in the application might have an inner loop that looks something like:

```
for(i=0;i<total;i++)
    if (time[i]>start && interesting([ID[i]]))
        combine(accum[],ID[i],rain[i],time[i]);
```

In this code fragment, `interesting` is a dynamically defined user-defined function that indicates which sensors the user is interested in. The user-defined `combine` function describes how the rainfall values should be grouped and accumulated/aggregated into the `accum` array. This query is asking for the aggregate rainfall for sensors that are `interesting`, over a time window between `start` and the current time. (These aggregates would be normalized at the end according to the `interesting` sensor count in each grouping region.)

Despite the relative compactness of this code fragment, there are several performance opportunities (and pitfalls) that might be

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447697

taken advantage of (or avoided) by the compiler that generates machine code from this loop. Performance diversity can be caused by variations in the data distribution and/or physical data ordering, influencing branch-related and cache-related stalls. Different algorithmic choices (reordering of operations, use of SIMD instructions) can improve or worsen performance. Details of these opportunities and pitfalls will be discussed at more length later in Section 3.

Our work builds on the dynamic query execution scheme pioneered by Vectorwise [55] (discussed in Section 2). Multiple plans are precompiled for a particular operation. As the operation progresses over a very large data set, performance information from the early stages of execution can be used to guide the choice of plan for later stages. Plan switching allows for robustness in the face of errors in query cost estimation, and also allows a dynamic change of plans if the data distribution changes within the dataset. Details of our adaptive code generation are given in Section 4.

Existing query optimization techniques for in-memory processing are limited in several ways: (a) they are not extensively used outside relational database management systems; (b) they are limited to a handful of relational operators, and do not cover access patterns or dynamically-defined functions found in other data-analysis scenarios; (c) they treat the underlying compiler as a black-box, with unpredictable performance depending on which compiler is used with which compiler settings; (d) they often bake-in design choices that may be appropriate for usage within a particular DBMS, but not for more general cases. We address these challenges by optimizing data-analysis style queries expressed as tight loops in a conventional imperative programming language.

We extend an open-source compiler (GraalVM compiler [65] and Truffle [64]) with both known and novel optimization techniques that can automatically be applied whenever the compiler identifies that a loop is time-consuming. GraalVM is an ecosystem and shared runtime offering performance advantages for a variety of programming languages [45]. Interpreted code is automatically transformed into compiled code when the system detects a performance hot-spot. The GraalVM Compiler is a dynamic just-in-time (JIT) compiler that performs sophisticated code analysis and optimization. The Truffle API allows programming languages to be combined in a shared runtime using an abstract syntax tree representation. Interpreted code is associated with nodes in the abstract syntax tree, and the Graal compiler automatically compiles the performance-critical parts of the code to speed up execution. Details of the Graal/Truffle implementation are provided in Section 5.

Integration into the compiler enables many applications to efficiently process large data sets. The system supports dynamic queries involving user-defined functions and arbitrary access patterns. Database-style and compiler optimizations co-exist, eliminating some of the mismatches that happen when the compiler is used as a black-box by a DBMS. The system tunes a variety of run-time execution parameters automatically, with minimal guidance from the programmer.

We evaluate our system using the TPC-H benchmark, weather visualization, and microbenchmark queries, over datasets with various kinds of ordering/clustering properties. The experimental evaluation (Section 6) shows that:

- Our system can dynamically respond to changes in the data distribution, choosing the best plan for the current data.
- Our system can invoke SIMD optimizations for code, even though they do not always improve performance. In our system, the SIMD version will be used if it is better, and the scalar version will be used otherwise.
- The system can select a small but representative set of plans that cover the search space well enough to respond to various parameter combinations that may not have been known at query compilation time.
- It is possible to dynamically achieve a balance between exploration (trying out a variety of plans) and exploitation (maximally employing the best plan).

2 BACKGROUND

2.1 Prior Work on Compiling Query Plans

Our work builds upon the dynamic query execution scheme developed as part of the Vectorwise system [55]. The Vectorwise implementers observed that query performance could vary significantly due to low-level performance effects. Different query plans might perform best under different regions of parameter space, yet the parameter values may not be known at compile time. Different compilers for the same programming language might give better or worse results, depending on the query. Data distribution effects (that may change as the system progresses through the data) may affect query performance, so that one plan is best for parts of the data, while another plan is best for other parts.

The Vectorwise team also observed that it is hard to estimate the cost function, not just because of the data distribution effects and parameter estimation inaccuracies mentioned above. Different run-time platforms may have different performance characteristics, such as the relative cost of a SIMD instruction to a scalar instruction or the relative impact of a branch misprediction. Further, the overlapping of various latencies (e.g., cache misses) makes it hard to identify their true impact on elapsed time. Rather than estimate the cost, Vectorwise chose to *measure* the actual cost.

In Vectorwise, data chunks of about 1000 rows are processed as a unit. A key innovation in Vectorwise is the analysis of the actual running time over recent chunks of data using several different candidate query plans in turn [55]. Each plan contributes to the final result, but might take more or less time depending on data and machine parameters. The plan that takes the least time is scheduled to run for an extended number of chunks. After that, all candidate plans are run again within a certain window to see if the data has changed to the point that a different plan is best. The best plan is then scheduled for an extended period, and the process repeats.

To summarize, the advantages of the approach pioneered by Vectorwise are: (a) optimization happens on the basis of actual time rather than predicted time, reducing the reliance on complex and potentially inaccurate cost modeling; (b) most of the execution will use the best plan among the candidates; (c) over time, as the data changes, the chosen plan can adapt to those changes. Despite these advantages, the Vectorwise approach has several limitations that we will discuss next.

2.2 Limitations of the Vectorwise Approach

The first and most obvious limitation of the Vectorwise approach is that the implementation effort has no wider impact beyond uses of

the Vectorwise system itself. It might be possible for a competing DBMS to mimic the implementation described by Vectorwise, but applications of the techniques beyond in-memory relational DBMSs are unclear. *In contrast, our approach embeds the optimization/execution decision making at the programming language level, making the techniques broadly applicable to a wide variety of applications.*

A second limitation is that the Vectorwise approach uses a few hand-crafted code fragments that cover only the essential DBMS operators. These code fragments are precompiled at DBMS build time. Code fragments with in-lined user-defined code are not considered. Access patterns in which there is interaction between consecutive rows are common in applications such as time-series analysis, but are essentially absent in a relational DBMS. *We compile code fragments at query-time, allowing user-defined code and arbitrary access patterns that might not match a handful of predefined templates.*

The paper describing the Vectorwise system describes how they used several different compilers, with different optimization settings, and observed varying performance results. The results were so unpredictable that they were forced to compile multiple variants of each code fragment: two compilers and two optimization settings would require four compiled code variants to cover all of the cases. The Vectorwise authors remarked that they resisted the temptation to investigate why the compilers had such different behaviors [55], presumably because they had no control to effect a change even of they could identify an inefficiency. *In our method, the compiler is not an external black box. Instead, because DB-style optimizations and traditional compiler optimizations happen in the same framework, we can control code generation. If the compiler is unsure whether an optimization helps or not, two variants of the code fragment could be generated internally, by the compiler itself.*

The Vectorwise system chooses somewhat arbitrary values for parameters such as the window size to run the current best plan, and the window size within which other candidate plans are run. *While these settings may have been adequate for the limited set of operators considered by Vectorwise, it is not clear that such choices would be optimal under the broader contexts considered in this paper. We investigate principled ways for setting such parameters, allowing them to vary based on performance feedback generated so far. Section 6 shows an experiment where the choice of window size matters.*

3 PERFORMANCE DIVERSITY AND REWRITING OPTIONS

Let us return to the loop introduced in Section 1, in which we first specialize and in-line the definition of `interesting`, and `combine`. The user has specified that an ID is interesting if its latitude is greater than 30, and has stated that the way to combine rainfall readings is to sum the `rain` amounts grouped by zipcode. `zip[id]` represents the zipcode where the sensor having identifier `id` is located, and `lat[id]` and `long[id]` represent the latitude and longitude of the sensor.

In database terminology this query applies two selection conditions, performs two foreign key joins to the `lat` and `zip` "tables", and performs a grouped `SUM` aggregate of the rainfall. We assume that `total` is very large, so that optimizing the loop is likely to have a big performance impact. The hot-spot compiler will be triggered relatively quickly to compile the code rather than continuing to run it in interpreted mode. We describe some of the performance-related choices that need to be made below.

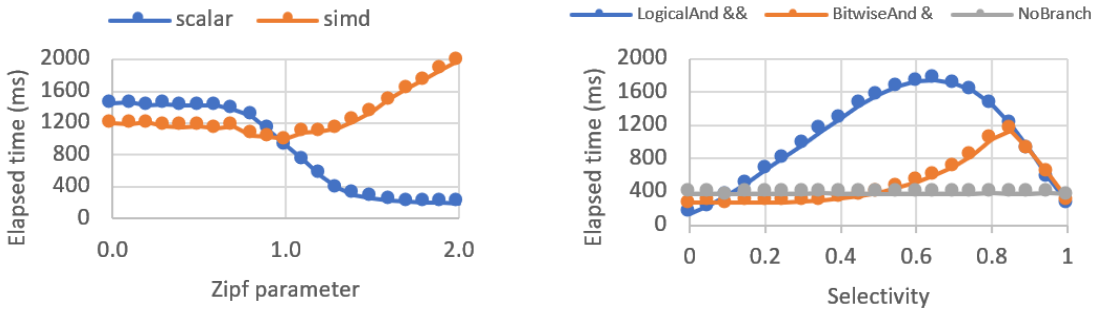
Condition Ordering and Non-Branching Plans. Selection condition order is important for in-memory query processing [54]. Branch misprediction effects contribute significantly to query processing costs. Among the plans considered are plans that avoid branches altogether by converting control dependencies to data dependencies. For example, the plan above might be rewritten as follows to avoid branches:

```
for(i=0;i<total;i++) {
    // & rather than &&; no branches
    test = (time[i]>start & lat[ID[i]] > 30);
    // -1 = 0xFFFFFFFF; -0 = 0
    mask = -test;
    // 0 mask means add 0, i.e., no-op
    accum[zip[ID[i]]] += (mask & rain[i]);
}
```

While branch-free code eliminates the branch misprediction overhead, it is not always the best choice. For example, if a condition is very selective, so that it fails most of the time, then executing the condition early is good because (a) it avoids unnecessary work for most tuples, and (b) conditions that fail most of the time are relatively well-predicted by modern processors. When several conditions are present, the best ordering of those conditions depends both on the selectivity of the condition and the cost of testing the condition [23, 54]. These kinds of alternative rewritings are used in the hand-generated templates of the Vectorwise system [55]. Our system automatically generates candidate plans at query-time using each kind of rewriting (details in later sections).

Cache Misses. Accesses to the arrays `time`, `ID` and `rain` are sequential, and prefetching is likely to be effective in minimizing cache latency for those accesses. `lat` and `zip` are accessed non-sequentially, and may generate cache misses whose latency may be significant (tens of cycles for an L2 miss, about 100 cycles for an L3 miss). These costs also influence the ordering of selections, since a cache miss might make a condition like `lat[ID[i]] > 30` expensive to test. Whether `lat[ID[i]] > 30` generates a cache miss on `lat` depends on: (a) How many IDs there are in total and how compactly they are allocated in the `lat` array (e.g., are sensor IDs re-used when a sensor is taken out of service?); (b) How many IDs are likely to be registering rain at the same time (depends on sensor placement and weather patterns); (c) How likely it is that a sensor that registers rain at time t also registers rain at time $t + 1$ (affects temporal locality, and depends on weather patterns). Given the complexity of predicting cache behavior, we circumvent the problem by considering a very limited number of scenarios. For example, we might just consider two extreme scenarios, one in which we expect an L3 cache miss and one in which we expect an L1 cache hit.

SIMD. SIMD instructions can be applied to both the conditions and actions of the code above. Let w be the number of SIMD lanes. The condition `lat[ID[i]] > 30` might be evaluated on w consecutive `i` values by (a) loading a SIMD register with w consecutive ID values; (b) using a SIMD gather instruction to look up w different addresses within the `lat` array; and (c) comparing the results with



(a) Varying skew

(b) Varying selectivity

Figure 1: Performance diversity

a SIMD register pre-loaded with w copies of the value 30. The resulting booleans can then be ANDed with other boolean conditions, or used as a mask for other actions.

The update of the `accum` array can similarly use SIMD gather operations to load the current running sums, SIMD add instructions to perform the updates, and SIMD scatter operations to write out the results. Special SIMD instructions detect conflicts (e.g., updates to a common memory address) across SIMD lanes and serialize them in the same sequence as the input.

SIMD processing has the potential to speed up processing if the workload is not memory-bound by using fewer instructions to do the same work. It is not always clear that SIMD optimization is desirable because (a) similarly to no-branch plans, it does the entire work even if the first condition would have led to a quick rejection; (b) under conditions of skew, the conflict resolution step of the SIMD scatters may dominate the cost, making the SIMD option slower than the scalar option. Rather than trying to estimate skew and determine whether the exact cost of the SIMD option is optimal for the current data, we simply generate SIMD plans as additional candidates to be considered at run-time.

Performance Diversity. Figure 1 illustrates two cases of performance diversity alluded to in the previous discussion. Figure 1(a) shows the performance of a grouped aggregation, where the grouping column may be skewed according to a Zipf factor shown on the x-axis. The SIMD code is faster than scalar code under low skew, but slower under high skew due to the high cost of conflict resolution as described above [67]. Scalar code is fastest at high skew because the grouping cardinality is small and so the aggregates fit in the L1 cache. Figure 1(b) shows three plans for a query having two selection conditions, using plans of the kind described in [54]. Each of the three plans is best in some selectivity range. Because the selectivity may not be known in advance, or may vary within the dataset, our approach will be to include multiple plans and to choose the best plan according to the recent performance history.

4 ADAPTIVE CODE GENERATION

So far we have suggested that we will be generating multiple plans, running each for chunks of data during a testing phase, and then selecting the fastest plan to run for an extended period. Unlike the Vectorwise system, where an arbitrary number of plans might be precompiled in advance, we aim to generate plans at query time. This choice allows for more general plans, including in-lined user-defined functions that are not known in advance. Nevertheless, this choice is challenging because it makes query compilation itself part of the observable response time. Our preliminary observations using the Graal compiler (Section 5) suggest that a plan can be compiled in tens of milliseconds. Thus, if we were performing a large scan taking several seconds, say, we could probably not afford to compile more than 10 plans. Beyond that, the overhead of compilation may outweigh the benefits of adaptive query processing/optimization.

4.1 On-Line Analysis

First, we optimize *abstractions* of the loop components. For example, the cost estimate for a SIMD computation may depend on the skew in the group-by values (Figure 1(a)). We may simply optimize under two abstracted conditions: no-skew and high-skew. As a second example, the cost estimate for a condition-testing plan may depend on the selectivity (Figure 1(b)) and cache-behavior of the data. Rather than estimating a selectivity for a condition, we *impose* a selectivity on that condition as a way of making sure we cover an appropriate subregion of the optimization space. A condition may be given selectivities that are "small," "medium," or "large" (say 0.05, 0.5, 0.95 respectively).

4.2 Off-Line Analysis

There is an implicit bias in our on-line analysis, because our relatively coarse abstractions of parameters may be far from either (a) the true parameters, or (b) the critical values of the parameters for which the choice of plans would change. We therefore supplement our on-line analysis with an off-line analysis for common query patterns. For example, we imagine that loops containing if-statements that test any number of conditions may be common in practice. We therefore perform a more detailed off-line analysis of c-condition loops for all c below some moderately large threshold (at least 10). Although this off-line analysis is expensive, it would happen once for a target hardware environment before the compiler is released, or during a calibration step when the compiler is installed. After the off-line analysis, the system stores the generated candidate plans as a summary to use for adaptive code generation online (Section 5).

For each c, we use a more fine-grained approach to compute a cost estimate of candidate plans for c conditions based on the cost

Table 1: Candidate plans

#	plans (exhaustive)	ratio	plans (local)	ratio
1	{ C0 & C1 & C2 }	9.77	IF (C0) { C1 & C2 }	12.64
2	IF (C0) { C1 & C2 } IF (C1 & C2) { C0 }	5.40	IF (C0) { C1 & C2 } IF (C1) { C0 & C2 }	12.07

3	IF (C0) { C1 & C2 }
	IF (C1) { C0 & C2 }
	IF (C2) { C0 & C1 }
		3.25	IF (C0) { C1 & C2 }
			IF (C1) { C0 & C2 }
			IF (C2) { C0 & C1 }
				3.25

4	{ C0 & C1 & C2 }
	IF (C0) { C1 & C2 }
	IF (C1) { C0 & C2 }
	IF (C2) { C0 & C1 }
		1.97	{ C0 & C1 & C2 }
			IF (C0) { C1 & C2 }
			IF (C1) { C0 & C2 }
			IF (C2) { C0 & C1 }
				1.97

5	{ C0 & C1 & C2 }
	IF (C0) { C1 & C2 }
	IF (C1 && C0) { C2 }
	IF (C1 & C2) { C0 }
	IF (C2 && C0) { C1 }
		1.79	{ C0 & C1 & C2 }
			IF (C0) { C1 & C2 }
			IF (C0 & C1) { C2 }
			IF (C1) { C0 & C2 }
			IF (C2) { C0 & C1 }
				1.97

formulas of [54]. For example, for 3 conditions, we try all 6 orders as well as all logical-and, bitwise-and, and no-branch plans. Since we do not know the selectivity and cost of each condition (and the cost of the body part) in advance before query execution, we develop a large number of configurations in an offline analysis. For every condition, we test 20 selectivities ranging multiplicatively from 0.0001 to 0.9999. We test 10 cost values from 1 to 1024 cycles, again multiplicatively. Then, for each of these 20x10 configurations, we compute the cost of all different plans [54].

We then compute a summary of the best plans to use during online exploration. Suppose we can afford to use k plans for exploration. Our metric for evaluating the quality of a set of k plans is based on the worst-case ratio of estimated performance across all configurations:

$$\max_{\{\text{configurations}\}} \left(\frac{\text{the best cost among the } k \text{ plans}}{\text{the best cost among all plans}} \right)$$

Then we would like to choose the set of k plans that minimizes this ratio. An exhaustive search would be too costly (exponential in k) and so we propose the following heuristic method.

(1) Every plan is considered as a valid candidate, and every configuration is mapped to the plan that minimizes its cost (which we record as the baseline cost for the configuration, to be used in the denominator of the formula above). Any plan that contains no configurations at this point is eliminated.
(2) While there are still too many plans, consider each plan P in turn as follows: (a) Map each configuration previously assigned to P to the next-best plan, and compute the ratio of the new estimated cost to the baseline cost. Record the highest cost ratio as the score for P. (b) Remove the plan with the lowest score, and re-assign its configurations to their next-best plans.

We eliminate plans with the lowest ratio because their elimination makes the smallest incremental difference to the overall ratio we are trying to minimize. In other words, the next-best plans are almost as good as the elimination candidate.

Table 1 shows how this algorithm performs for 3 conditions ($c = 3$) and up to 5 plans ($1 \leq k \leq 5$). For comparison, we also show the results of an exhaustive search. In general, the best set of $k-1$ plans may not be a subset of the best set of k candidate plans, but our heuristic algorithm does choose $k-1$ plans from among the best k plans. We observe that the heuristic performs reasonably well when $k \geq 3$, which is likely in our application domain.

For small k, an exhaustive search will be feasible, and it does not miss the best plans that the above heuristic could prune. Therefore we use a hybrid approach: Generate the best 10 plans using the heuristic, and then search exhaustively among them for the best pair of plans. This aproach is more accurate for small number of candidate plans.

We used the *maximum* performance ratio as our heuristic function, but we could alternatively have used the *average* performance ratio. We argue that the average can be biased depending on how the selectivity and cost values are chosen. For example, they would give extra weight to the regions of parameter space that were more heavily sampled. In contrast, the max ratio is relatively stable, and focuses the optimization on the part of the parameter space where it matters most.

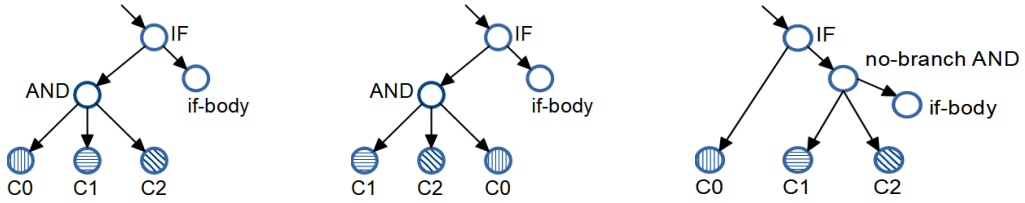
Table 1 shows that there is a diminishing return in reducing the max ratio metric as we choose more candidate plans. During online execution, the best plan among the candidate plans is chosen. Assuming that there is enough data for exploitation (so the exploration cost is negligible), then it is in theory better to choose from more candidate plans, but the marginal benefit is decreasing (as is the metric). As we demonstrate in the experiments, the performance stabilizes as we increase the value of k. In practice, a reasonable heuristic for k conditions is to use at least k candidate plans so that every condition can be the first condition in some plan.

4.3 Measuring Execution

We follow the Vectorwise approach by measuring actual times and choosing plans based on their recent history of execution times. The Graal/Truffle system already instruments interpreted code with counters to observe events like a branch being taken. When the interpreted code is identified as a hot-spot and compiled, that information is used to inform the subsequent compilation phase. The counter instrumentation is omitted from the compiled code to minimize overhead.

For the execution of compiled code, we divide the entire execution into a series of alternative exploration and exploitation periods. During an exploration period, a number of candidate plans is tested over input chunks and compared with the execution times. In the following exploitation period, the best plan is maximally employed over a larger number of chunks. We keep a recent history of chunk execution performance, so that the system can react to changes by comparing the current execution with previous executions. Two heuristics are used for dynamically setting parameters:

- **Dynamic exploitation** (DE). For consecutive exploration periods, if the best plan does not change, then it suggests that the data is behaving consistently, so we double the size of the exploitation period; otherwise, the data distribution is likely to have changed during the two explorations, so we reduce the size to half of the original exploitation period.



(a) AND plan **(b) Reorder** **(c) No-branch**

Figure 2: Rewriting of a conditional AST

- **Early exploration** (EE). When we observe that the chunk takes significantly longer to execute (more than double the average of recent chunks), it is a strong indication that the underlying data has changed, so we start exploration using additional plans starting from the next chunk.

In practice, combining these two heuristics works well for our experimental datasets (Section 6.2).

5 IMPLEMENTATION

We use the Truffle language implementation framework [64] to develop the adaptive execution framework. Truffle is an open-source library that simplifies the development of language execution engines and data processing engines using self-optimizing abstract syntax trees (ASTs) in the GraalVM ecosystem. Each node in the AST represents an operation (e.g., a comparison, an evaluation of an AND condition, an arithmetic computation, etc.) that is compiled to machine code by the Graal compiler. During the execution, an AST node can make use of runtime information and change its internals to specialized versions that have better performance. Node rewriting and JIT compilation are automatically handled by the Graal compiler.

In this paper, we focus on JavaScript programs with a for-loop like the example in Section 1. Users can write a *pragma* directly above the for-loop they wish to perform adaptive execution on:

```
var input0 = ...      // initialize data arrays
var input1 = ...
var count = 0;

"adaptive execution"; // adaptive execution pragma
for (i=0; i<1000000000; ++i)
    if (input0[i]<20 && input1[i]<50)
        count++;
```

By using the pragma, the user is (a) certifying that the predicates in the if-statement can be reordered, and (b) hinting that adaptive execution should be applied to the for-loop.

5.1 Preprocessing

Upon execution of the JavaScript program written by the user, a custom script first rewrites the program source code to use the Polyglot API. Polyglot allows different languages implemented with Truffle to interoperate with each other. In our implementation, we use Polyglot to access variables in JavaScript, and make the following changes to the source code: (1) The for-loop itself is transformed into a string; (2) Values of all variables that are used in the for-loop, but defined outside of the for-loop, are stored in a dictionary; (3) The variable dictionary and the for-loop string are passed to the code generation framework via the Polyglot API.

To control the adaptive code generation, we implement a set of AST nodes extending from Truffle `Nodes`, including value nodes (e.g., constants), arithmetic nodes (e.g., `Addition`), and condition nodes (e.g., `LessThan`). When the rewritten source code is executed and the adaptive execution framework is invoked, control is handed over to the root node, a special Truffle AST node that handles the execution of the loop and measures the performance. We use a custom parser built with ANTLR to parse the for-loop string into the Truffle expression nodes we implemented, and generate multiple ASTs representing the candidate plans according to the summary obtained from offline analysis (Section 4.2). The variable values stored in the dictionary are written to the procedure stack, so that they can be accessed and modified during the adaptive execution.

Under the root node of the loop, a `TopLevelCondition` node represents the if-statement. For conjunctive conditions (`AndCondition`), a candidate plan specifies the ordering of the conditions as well as a mode indicating how the conditions are computed and combined together (LogicalAnd, BitwiseAnd, or NoBranch). The ordering and the node properties are stored as internal variables of an `AndCondition`. The body part of the if-statement (true branch) is a generic AST node if all conditions have been evaluated. If there are remaining conditions to be evaluated as no-branch conditions, then the body part is rewritten to an `AndCondition` node with NoBranch mode. The body also uses a mask to determine whether the result is written to output. Multiple assignment statements are permitted in the body.

Depending on the number of conditions (i.e., the structure of the code), the root node chooses from a summary with matching conditions a set of candidate plans. For each candidate plan, the root node constructs an AST as shown in Figure 2. An `AndCondition` node has conditions as its child nodes, which are basic conditions like LessThan comparisons. Figure 2 shows three example plans with the same semantics. By reordering the conditions (C0, C1 and C2), the AST in Figure 2(a) is rewritten to Figure 2(b) and thus executed differently. Either logical or bitwise AND can be used depending on the mode set in the `AndCondition`. If a no-branch plan is used then an `AndCondition` with the no-branch mode is used to rewrite the plan into Figure 2(c), where only condition C0 is executed with a branching if-condition. We then invoke the Graal compiler backend to compile the AST into callable machine code.

When there are no if-conditions as in the example in Section 6.1, then the body part is just an AST representing the assignment

Table 2: Time breakdown (s), 10^9 rows, median of 10 runs

	Adaptive	&&	&	reverse &&
Preprocess	0.10	0.11	0.11	0.11
Execute (interpreted)	0.22	0.07	0.07	0.07
Compile	0.81	0.55	0.55	0.55
Execute (compiled)	6.28	11.49	5.94	11.47
Total	7.44	12.22	6.67	12.19

and arithmetic computations. Additionally, we support SIMD code generation for basic arithmetic computations. The implementation extends the Graal compiler to add intrinsics using AVX-512 instructions. For the example in Section 6.1, we implement the computation and conflict resolution in SIMD as a compiler directive. Based on a template, the root node recognizes the code structure written in scalar code, and generates the corresponding body node using the specific compiler directive. The candidate plans are then the scalar and SIMD versions of the body node.

After the above preprocessing, the root node triggers the compilation of the candidate ASTs. Note that the program has to run in the interpreted mode for a very short time before the compilation is triggered. Then during an exploration period, the root node tries the compiled candidate plans and measures the actual execution times of a chunk. For each plan, we run over 2 chunks, and measure the time of the second chunk, to overcome instabilities of the first chunk measurement. The chunk size is set to 1000 tuples so that we can amortize the overhead of time measurement and still get a rank of the plans. The best plan is then used for the longer exploitation period. When dynamic exploitation (Section 4.3) is used, we also track 10 recent chunk execution times to enable the heuristic.

5.2 Compiler Overheads

To measure the overheads of compilation itself, we measured the time taken for an end-to-end compilation of three plans followed by an execution of a loop over 10^9 tuples. We compare with the time taken by the unmodified Truffle/Graal compiler on each plan individually. We used the G1 garbage collector for Truffle/Graal with default settings, and performed the experiments on a Xeon E5620 machine. For this experiment, one of the three plans (the & plan) is optimal for the whole dataset and so compiling this plan directly represents a baseline for the adaptive technique. The results in Table 2 show that the compilation overhead for adaptive execution is small relative to the cost for compiling the optimal plan without adaptive execution. The performance of the non-optimal plans is significantly worse than adaptive execution. Adaptive execution is thus robust with respect to how a programmer might have initially coded the conditional expression test.

Compilation performance improves as we compile more plans as shown in Figure 3, primarily because the compiler itself is just-in-time compiled dynamically during execution. Based on the results in Figure 3 one could expect to reduce the compilation overhead from 0.81 seconds to 0.23 seconds if one were to precompile the compiler itself, as in `libgraal` [59]. The small spike in performance for the seventh plan is caused by the compiler's invocation of garbage collection.

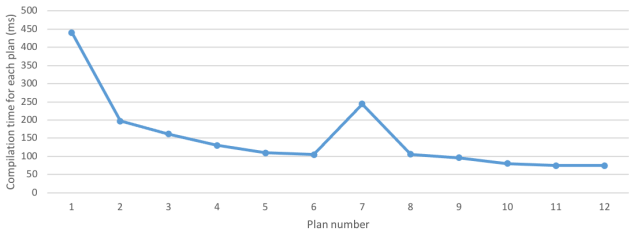


Figure 3: Compiling a sequence of plans

6 EXPERIMENTAL EVALUATION

In Sections 6.1–3, we conducted experiments on a Linux server with a 2.5 GHz Xeon Platinum 8175M processor. In Section 6.4, we demonstrate support for a visualization application using the adaptive execution approach, running on a laptop with an Intel Core i7-1065G7 processor. The execution of the query program uses a single thread, processing in-memory datasets stored as arrays of data. Given the relatively low, fixed overhead of compilation, we focus in this section on the main loop execution time using compiled plans.

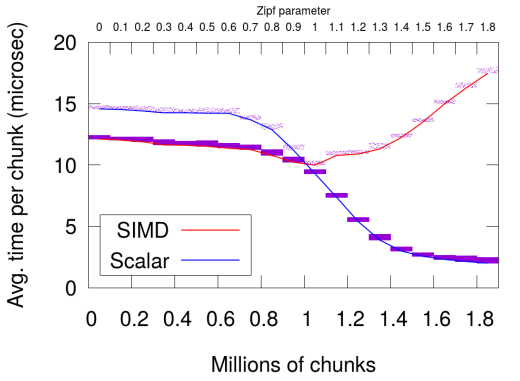
6.1 Microbenchmark on Skewed Data using SIMD

We present one set of experiments to show how our system can adapt to data distributions, and to illustrate how parameters such as the exploration window size might be worth tuning for optimal performance. Our baseline query has the form:

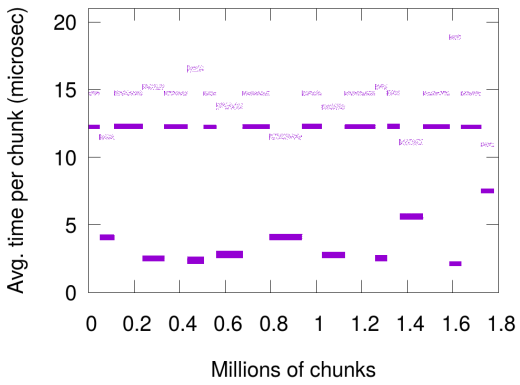
```
for(i=0;i<n;i++)
    output[data[i]] += compute(i,data[i]);
```

The `compute` function involves 3 logical shifts, 3 exclusive-ors, and two integer multiplications, all of which can be performed in a data-parallel fashion using SIMD instructions. `compute` is in-lined to avoid the function call overhead. An important aspect of this loop is the distribution of the `data[i]` values. A narrow distribution will lead to better cache locality in the `output` array, but potential conflicts to resolve common outputs from different SIMD lanes. A broad distribution will have worse cache locality, but will be mostly conflict-free, as discussed in Section 3. We model skew in the `data[i]` values by using a suitable Zipf distribution with a z parameter between 0 (uniform) and 1.8 (highly skewed). $z \approx 1$ is a common value for real-world skewed data. The data is divided into 1.9 million chunks of size $1,000$ (1.9 billion records).

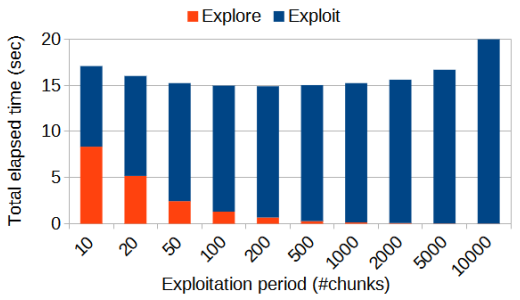
Figure 4(a) shows the performance of the compiled code when the data distribution becomes more skewed as the index increases: the first 100,000 chunks have $z = 0$, the next 100,000 chunks have $z = 0.1$ and z is subsequently incremented by 0.1 each 100,000 chunks. There are two implementations for the loop: (i) a standard scalar implementation whose performance as a function of z is shown in blue; and (ii) a SIMD implementation with conflict detection/resolution, whose performance is shown in red. From these curves alone it is apparent that SIMD beats scalar for small z, because it can parallelize the work of the `compute` function. However, for large z, the SIMD algorithm becomes an order of magnitude worse than the scalar code due to the need for conflict resolution.



(a) z increasing



(b) z varying



(c) Effect of exploitation period

Figure 4: Performance on skewed data with SIMD

The purple dots in Figure 4(a) show the performance of the chosen algorithm for each chunk. There is high density in some regions leading to what appears to be solid coloring. The more diffuse dots are where alternative plans are run to obtain estimates of their performance, to see if it is worth switching plans. Figure 4(a) demonstrates that the active plan chosen tracks the better of the two plans. Figure 4(b) shows a variant in which the same loop is run over data whose Zipf parameter alternates between uniform and a randomly chosen z value at unpredictable points in the data set. This kind of data is what we might expect in our rainfall example: steady rain over a wide region might generate uniform-looking data, whereas a sequence of local storms might generate regions of data skew. A close examination of Figure 4(b) shows that at the beginning of each skewed region, there is a small period during which the inferior plan is being run. The system has not yet reached the next window where it re-evaluates plans; it continues executing the same plan until that happens.

To understand the impact of the length of the exploitation periods, we ran several experiments with different exploitation period sizes. Figure 4(c) shows the elapsed time spent in exploration and exploitation mode separately. When the exploitation period is too small, we waste time running suboptimal plans too often: a suboptimal plan that is 5X worse than optimal and run 3% of the time will constitute a 12% overhead. When the period is too large, we do not notice a change in the data distribution until we have been running a suboptimal plan for a while. For this example, the best intermediate value for the exploitation period is around 200 chunks, and the exploration mode takes 4.3% of the time.

6.2 Microbenchmark with Different Selectivities

The previous subsection demonstrates how to choose between two code generation options (scalar and SIMD) adaptively. In this subsection, we study how to choose among various plans of conjunctive conditions. The query used in this set of experiments has the form:

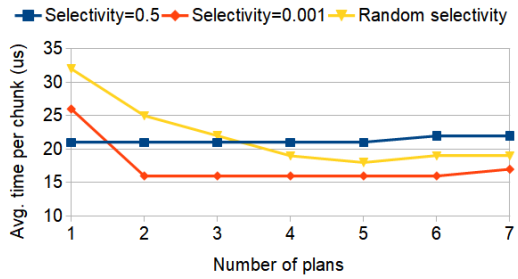
```
for (i=0; i<n; ++i)
    if (a[i]<A && b[i]<B && c[i]<C)
        sum += compute(input[i]);
```

In this program, we have three range predicates over the input data arrays a, b, and c. For the microbenchmark, we generate random numbers in the a, b, c arrays and control the selectivity of three tests by setting the corresponding A, B, C constants. As described in Section 4, there are many different orderings and plans including the non-branching plans to be considered.

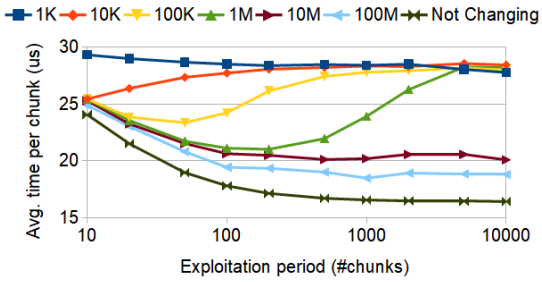
Figure 5(a) shows the average running time per chunk (in microseconds) on three different datasets, using a varying number of plans. Two of the datasets have a fixed selectivity: 0.5 and 0.001 for all three conditions. The third dataset sets random selectivities for each condition every 1M tuples.

For the dataset with 0.5 selectivity, the first plan from the summary generated by offline analysis is the no-branch plan, and it is always chosen as the best plan for this dataset. Adding more plans does not improve the performance. For the dataset with selectivity 0.001, the no-branch plan is not the best. If we use more than one candidate plan during the exploration period, then the candidate plans include the plan `if (a[i] < A){...}` (or other symmetrical plans), so it uses this plan as the best plan and reduces the average running time. For the changing data, we find that after 5–7 plans, the running time becomes stable, suggesting a suitable number of plans to use during online exploitation.

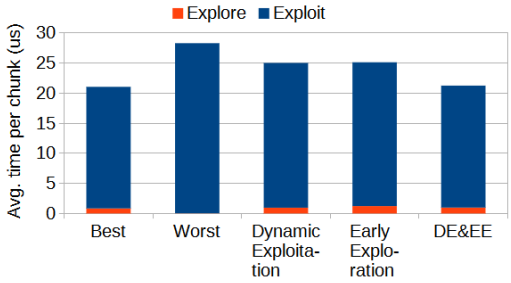
By default we use 200 chunks as the exploitation period. However, the underlying data distribution may change more or less rapidly than what we can detect. To study the effect of exploitation period, we generate datasets that change the selectivity (and thus the best plan) randomly. Using 5 plans, Figure 5(b) shows the average running time per chunk on the 7 different datasets, using a varying exploitation period (x-axis: the unit is the number of



(a) Varying plans

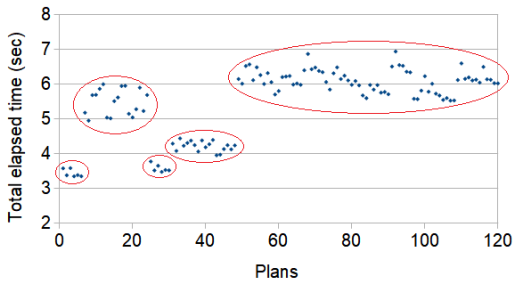


(b) Varying exploitation periods

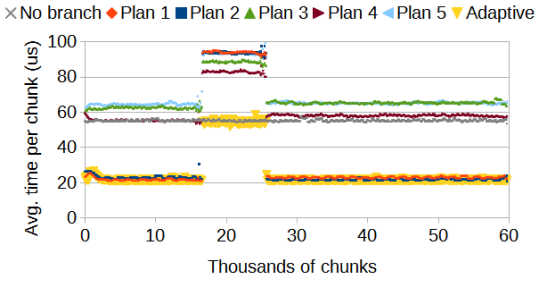


(c) Dynamic parameters

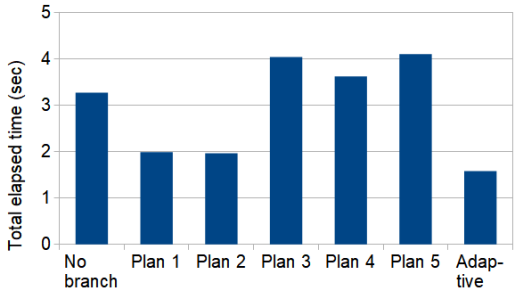
Figure 5: Performance with different selectivities



(a) Different plans on unsorted data



(b) Profiling on sorted data



(c) Performance on sorted data

Figure 6: Performance on TPC-H Q6

chunks, chunk size is 1000 tuples). The selectivity of a dataset is changing randomly every 1K (10K, 100K, ...) tuples, and the best exploitation period also changes correspondingly.

Figure 5(c) shows that the dynamic heuristics of Section 4.3 are able to achieve the performance of the best exploitation period (on the 1M dataset). Using one heuristic alone can avoid the worst case exploitation period, in Figure 5(b), and using both heuristics together we can achieve similar performance of the best exploitation period found.

6.3 TPC-H Queries

We now show results using code that implements TPC-H queries Q6 and Q19 [1]. We chose those queries because they have interesting condition structures that might benefit from our approach.

6.3.1 Query 6. Query 6 quantifies the amount of revenue increase that would have resulted from eliminating certain companywide discounts in a given percentage range in a given year. The query is written in a javascript program as an if-statement with five different conditions (range predicates).

```
for (i=0; i<N; ++i)
  if (shipdate[i] >= DATE_MIN &&
      shipdate[i] < DATE_MAX &&
      discount[i] >= DISCOUNT_MIN &&
      discount[i] <= DISCOUNT_MAX &&
      quantity[i] < QUANTITY)
        sum += price[i]*discount[i];
```

Figure 6 shows the performance of TPC-H query Q6. Figure 6(a) shows the results on unsorted data, as generated by the benchmark data generator. The running times are clustered by the first condition used during evaluation. The two best clusters correspond to the two range predicates on the shipdate column. From left to right, the clusters are:

- 6 points: DateMin, DateMax as the first two conditions
- 18 points: DateMin first, other conditions second

- 6 points: DateMax, DateMin as the first two conditions
- 18 points: DateMax first, other conditions second
- remainder: neither DateMin nor DateMax first

Since the first condition has the most impact on performance, the compiler uses the following ordering of predicates in the six candidate plans for adaptive execution:

(1) DateMin, DateMax, DiscountMin, DiscountMax, Quantity
(2) DateMax, DateMin, DiscountMin, DiscountMax, Quantity
(3) DiscountMin, DateMin, DateMax, DiscountMax, Quantity
(4) DiscountMax, DateMin, DateMax, DiscountMin, Quantity
(5) Quantity, DateMin, DateMax, DiscountMin, DiscountMax
(6) no-branch plan (order unimportant)

Each of the first five plans has a different first condition to be evaluated. Under the adaptive execution, the compiler automatically chooses the best plan to process most of the data, no matter how the program is written.

Figure 6(b) shows the results on data sorted by shipdate, which is likely to be the typical case in real world. We show the performance of each of the six plans, as well as the adaptive execution results. In this figure, we plot the average running time per chunk for every 200 chunks of input data in the exploitation period. Because the data is sorted by the shipdate column, there is a discontinuity at around 20,000 input chunks, corresponding to the lower bound `DATE_MIN` specified in the query. Before the discontinuity, the adaptive execution chooses Plan 1; during the immediately following period, it chooses the no-branch plan, because evaluating the Date conditions is extra work (they always succeed) and Plans 1 and 2 become the most expensive; after the data crosses the `DATE_MAX` threshold, the system chooses Plan 2.

Figure 6(c) shows the total running time on the sorted data for different plans. The compiled code automatically chooses the best variant among the six candidate plans, and the total execution time of the adaptive method is reduced compared with any single fixed plan.

6.3.2 Query 19. Query 19 reports the gross discounted revenue attributed to the sale of selected parts handled in a particular manner. The query's where clause is a disjunction of three conjunctions. Each of the three conjunctions has the same structure of predicates but the predicates have different parameter values. For this experiment, we preprocessed the text data so that the parameters are numetic values supported by our current implementation. We manually implemented adaptive execution for this query because our current full compilation pipeline currently handles only conjunctive expressions. Written as a JavaScript program, the foreign key join is executed as an index lookup into the referencing array.

```
for (i=0; i<N; ++i)
  if ((brand[partkey[i]] == BRAND1 &&
    container[partkey[i]] == CONTAINER1 &&
    quantity[i] >= QUANTITY1 &&
    quantity[i] <= QUANTITY1 + 10 &&
    psize[partkey[i]] <= SIZE1 &&
    shipmode[i] == SHIPMODE1 &&
    shipinstruct[i] == SHIPINSTRUCT1) ||  // Conj1
    (...) ||                              // Conj2
    (...))                                // Conj3
      sum += price[i] * (1-discount[i]);
```

Table 3: TPC-H Q19 time (us) on unsorted data

Plan (first condition)	Conj1	Conj2	Conj3
BRAND	**2767167**	2721858	2513244
CONTAINER	2820004	3051187	2902889
QUANTITY MIN	2968445	3611849	2503752
QUANTITY MAX	3498422	4088774	3975201
SIZE	3549345	4767787	3360388
SHIPMODE	3125625	**2654871**	**2409472**
SHIPINSTRUCT	3826366	2665140	2425727

For each of the seven predicates in Conj1, we could include at least one plan that checks the predicate first, for a total of seven plans. The same observation holds for Conj2 and Conj3. Since the conjunctions are quite selective, no-branch plans for evaluating the conjunction are excluded because they are likely to perform badly. A naive application of our approach would then need to generate 7^3 combined plans in order to cover all of the important cases.

Instead, we observe that because the conjunctions are relatively selective, and combined by disjunction, all of the conjunctions are likely to be executed for most rows. In other words, it is unlikely that a positive result from testing one of the conditions would be effective at short-circuiting the evaluation to avoid the other conditions. (We also verified that the running time of all 6 orders of the three conjunctions perform roughly the same.) If all three conjunctions are going to be executed almost all of the time anyway, we should optimize them independently. As a result we get $7 * 3 = 21$ plans rather than 7^3 plans. For this particular query, the three conjunctions have the same structure, and so 7 plans (with three instances of each) would suffice. However, the compiler does not know that the conjunctions have similar structure, and so it cannot share plans in this way. Instead of one exploration period, we now use three exploration periods separately for each of the conjunctions. In each exploration period, we select the best plan for one of the conjunctions.

Table 3 shows the running time on unsorted data. The table shows the first predicate of the plan for each of the three conjunctions, when the other two conjunctions each uses BRAND as the first predicate. In adaptive execution, Conj1 chooses the BRAND plan, while Conj2 and Conj3 each choose the SHIPMODE plan. As a result the adaptive execution takes about 2.4 seconds to compute the revenue loss no matter how the program is written (i.e., the ordering of the conditions by the programmer does not matter due to adaptive execution). This performance is about twice as good as the worst plan in Table 3, which is probably not the worst plan overall. This experiment shows that the advantage of our approach includes robust performance even for complex conditions involving conjunctions and disjunctions.

When the data is sorted by SHIPMODE, there is a region of data where the equality predicate on SHIPMODE is always satisfied. We find that each conjunction either uses the SHIPMODE plan to quickly filter out the unqualified data, or when the SHIPMODE equality predicate is satisfied, uses the BRAND plan since it is the most selective and thus has the best performance. Figure 7 shows the runtime profiling of the BRAND plan, SHIPMODE plan, and the adaptive execution. The other plans behave similarly to

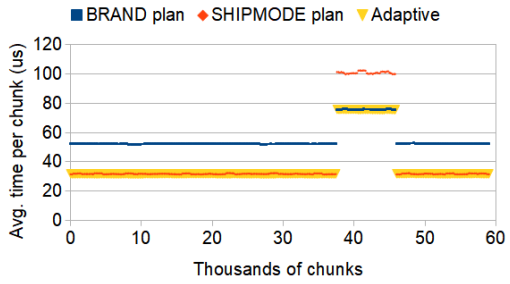


Figure 7: Performance on Q19 (sorted)

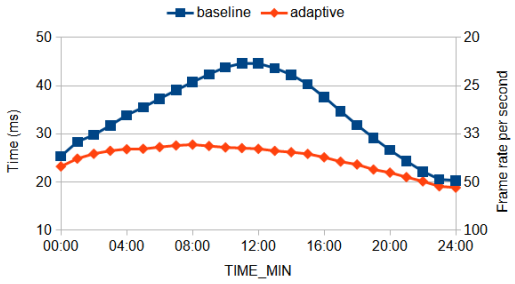


Figure 8: Performance of varying TIME_MIN

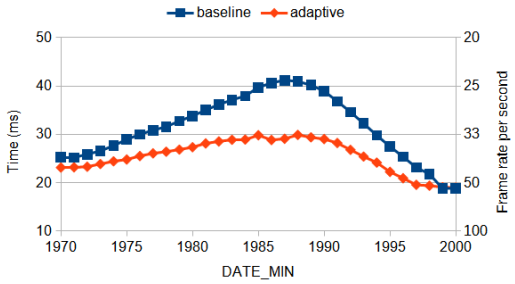


Figure 9: Performance of varying DATE_MIN

the BRAND plan, but they are slower; for clarity, their profiles are omitted in the figure. The profile of the adaptive execution overlaps with the BRAND plan when the SHIPMODE test is true, demonstrating that the execution switches to a different plan when the underlying data is changed. As a result, the adaptive plan takes 2.25 seconds to complete the computation, compared with a fixed BRAND plan taking 2.81 seconds and a fixed SHIPMODE plan taking 2.46 seconds.

6.4 Visualization Application

As motivated earlier, we imagine a hypothetical tool written in a JavaScript-like language for visualizing weather data. The tool allows users to choose a range of dates and times, and to select a bounding box on a map based on a range of latitudes and longitudes. The tool also computes some aggregation results using the data points that fit into these ranges. As the user adjusts a slider to change the parameter values, the program dynamically recomputes the query results, and the user interface interactively refreshes the screen to display new results. If the computation is slow for some parameter values, the display frame rate drops and there could be perceptible jitter as the user operates the slider in the graphical user interface.

We downloaded a real climate dataset containing historical 15-minute precipitation observations for selected U.S. stations[1]. The dataset has 18.2 million data points for the precipitation amount at 34354 stations across the U.S, for the period from 1971 to 1998. The precipitation data (date, time, amount) and the station data (latitude, longitude) are stored separately, and we load them into memory as separate arrays of data, keyed by the station id. Some of the data points are marked as invalid in the dataset. The entire dataset is about 3GB, so one is able to process them on a laptop computer instead of a server as in previous experiments. As an example, we compute the total amount of precipitation using the following program. It has nine conditions and computes a sum of the precipitation for measurements that satisfy all the conditions. Since station locations are stored separately, an indirect lookup is used to check the latitude and longitude of station locations.

For each of the nine predicates, one of the candidate plans is included to check that predicate first. Since the dataset is ordered by station, data points satisfying the latitude and longitude conditions are clustered in multiple regions. For different selectivities of the predicates, the best plan changes during an adaptive execution.

[1]https://www.ncdc.noaa.gov/cdo-web/datasets

```
for (i=0; i<N; ++i)
  if (data[i] != INVALID_DATA &&
      time[i] >= TIME_MIN &&
      time[i] < TIME_MAX &&
      date[i] >= DATE_MIN &&
      date[i] < DATE_MAX &&
      latitude[station[i]] >= LAT_MIN &&
      latitude[station[i]] < LAT_MAX &&
      longitude[station[i]] >= LON_MIN &&
      longitude[station[i]] < LON_MAX)
        sum += data[i];
```

Figure 8 shows the execution time of the program (left y-axis) and the derived frames per second (FPS, right y-axis) as the user controls a slider to vary the TIME_MIN value from 00:00 to 24:00. We compare the baseline method using the example program shown above with the adaptive execution approach. As the TIME_MIN varies to around 12:00, the selectivity of condition `time[i] >= TIME_MIN` becomes closer to 0.5, incurring an expensive branch misprediction overhead. Therefore, the baseline method has an execution time of 45 milliseconds and the framerate drops to below 25 FPS. The adaptive execution method, however, is able to switch to a different plan that checks TIME_MIN conditions later after checking other conditions, so its execution time is at most 27 milliseconds and the frame rate stays above 36 FPS. The difference in FPS is over 1.5x, and is well within the limits of human visual perception [30].

Figure 9 shows the running time and visualization frame rates as the user slides the DATE_MIN value from 1970-01-01 to 2000-01-01. The results are similar to the former case, and the difference in

FPS is up to 1.4x. Changing TIME_MAX or DATE_MAX conditions result in similar observations.

Since the dataset is not globally ordered by any single dimension of the nine conditions, during the adaptive execution of the program in the experiments of Figures 8 and 9, seven out of the nine candidate plans were exploited at least once. This observation emphasizes the need for a diversity of plans to handle runtime configurations that are difficult to predict, and the ability of our system to dynamically choose an appropriate plan.

7 RELATED WORK

Column-oriented execution [42] and cache-conscious operators [43] were proposed before the advent of multi-core CPUs. Block-at-a-time execution [8] and query-dependent code generation [21, 38, 44] are both state-of-the-art designs for analytical query engines [33]. The present work has features from both block-at-a-time execution and query-dependent code generation.

SIMD optimizations have been applied to a variety of database operators including joins [3, 4, 6, 27, 34, 58], sorting [10, 26, 50, 56], scans [69] and compression [40, 52, 63]. Advanced SIMD optimizations [49, 51] include non-linear-access operators. SIMD optimizations work best when data is cache-resident [68], the there are trade-offs between scalar and SIMD code as we demonstrated in Section 3.

Adaptive query processing aims to refine a query plan at runtime on the basis of statistics gathered at intermediate stages of the query computation [2, 14]. Multiple sub-plans could be compiled into a query, with a choice to be determined based on partial computations such as the size of an intermediate table. Alternatively, when a departure from the predicted behavior occurs, another round of query optimization could be performed at run-time. Early work on this topic instrumented query code with counters to gather statistics that inform such choices [13, 28]. More recent work using in-memory databases uses hardware performance counters to gather such statistics without any performance overhead [66].

We use a limited number of query plans based on an analysis of regions of parameter space. The Picasso database query optimizer visualizer allows one to visually inspect optimal plan choices for different regions of the parameter space [15, 22]. Our choice of a small number of plans is analogous to how Picasso would create a "reduced diagram" with a bounded reduction in overall performance. Empirically, the authors find that ten plans is almost always sufficient to cover the parameter space with at most a 20% degradation in the plan cost at any point in the space [15]. PlanBouquets [17] incrementally discovers actual selectivity at runtime in order to identify appropriate plan to execute, and recent work [31] has improved its significant compile-time overheads. Our plans are likely to be simpler than the ones considered by Picasso, so fewer then ten plans may typically be sufficient.

To deal with arbitrary user defined functions, [12] compiles a high-level query workflow into a distributed program. UDFs are compiled with LLVM into intermediate representations and then linked with the workflow program into binary executables. A different approach proposed recently is to compile UDFs into plain SQL queries [16, 24], where arbitrary control flows are translated into recursive expressions.

Database and programming language compilers have a common goal, namely to generate efficient machine code for queries/programs written in a high-level language. Recent query compilers resemble programming-language compilers, sharing some of the low-level infrastructure such as LLVM [12, 44]. The programming language community has built hot-spot compilers [46] that initially interpret (and profile) code sections. When the interpreter determines that the code section is a hot-spot, it pauses, compiles the code section in real time, and executes the remainder of the code section using the compiled code. This choice balances compilation and execution time, and similar innovations have recently been described for database query compilation [36]. While database compilers have adopted programming-language innovations such as LLVM and hot-spot compilation, our method shows that there is also an opportunity for technology transfer in the opposite direction.

Our system extends the Truffle framework [64] and the Graal compiler [65]. Using Graal as the host compiler, Truffle is particularly well-suited for languages with very dynamic semantics and whose execution depends heavily on the size, layout and contents of the input data. Truffle offers numerous primitives for collecting information about the observed data types and program behavior. Additionally, so-called assumptions allow for non-local optimizations where the point that uses optimized code based on a specific assumption is only loosely connected to the points that potentially invalidate this assumption. Leveraging this speculative just-in-time compilation based on implicit schemas that are discovered at runtime, Truffle has also been used to develop efficient parsers for JSON and CSV data [9], and to accelerate data de-serialization [57]. The existing profiling and assumption mechanism in Truffle are based on heuristics; they are local, behavior-centric, and strictly stabilizing (always moving towards the most generic version). This paper extends them with a dynamic mechanism, directly observing the actual performance of different but semantically equal algorithms.

8 CONCLUSIONS

We studied optimization techniques for data-analysis style queries expressed as tight loops in a conventional imperative programming language. Since the data distribution often strongly affects query performance, it is important to make the code generation and execution adaptive to the underlying data. To adapt to this performance diversity, we built upon an open-source compiler to generate code that efficiently processes large data sets with varying data distributions and predicate selectivities. By using a learning framework with alternative exploration and exploitation periods, we enabled code generation using different plans and SIMD options. We showed that the system could tune run-time execution parameters automatically, with minimal guidance from the programmer. As a result, we achieved robust query performance in both microbenchmark and TPC-H queries. When the underlying data changes, the adaptive code generation and execution can in fact achieve better performance.

ACKNOWLEDGMENTS

This research was supported in part by a gift to Columbia University from Oracle Corp, and by NSF grant IIS-2008295.

REFERENCES

[1] [n.d.]. The TPC-H Benchmark. http://www.tpc.org/tpch.

[2] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 261–272.

[3] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Ozsu. 2013. MultiCore, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB* 7, 1 (Sept. 2013), 85–96.

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Ozsu. 2013. Main-memory Hash Joins on Multi-core CPUs: Tuning to the Underlying Hardware. In *ICDE*. 362–373.

[5] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. 2012. Business Analytics in (a) Blink. *IEEE Data Eng. Bull.* 35, 1 (2012), 9–14.

[6] Spyros Blanas, Yinan Li, and Jignesh Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *SIGMOD*. 37–48.

[7] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51, 12 (Dec. 2008), 77–85. https://doi.org/10.1145/1409360.1409380

[8] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*.

[9] Daniele Bonetta and Matthias Brantner. 2017. FAD.js: fast JSON data access using JIT-based speculative optimizations. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1778–1789.

[10] Jatin Chhugani et al. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *VLDB*. 1313–1324.

[11] Confluent Inc. 2019. Streaming SQL for Apache Kafka. https://www.confluent.io/product/ksql.

[12] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *PVLDB* 8, 12 (2015), 1466–1477.

[13] Amol Deshpande and Joseph M. Hellerstein. 2004. Lifting the Burden of History from Adaptive Query Processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 948–959. http://dl.acm.org/citation.cfm?id=1316689.1316771

[14] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. *Adaptive query processing*. Now Publishers Inc.

[15] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. 2007. On the Production of Anorexic Plan Diagrams. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. 1081–1092. http://www.vldb.org/conf/2007/papers/research/p1081-d.pdf

[16] Christian Duta, Denis Hirn, and Torsten Grust. 2019. Compiling PL/SQL Away. *arXiv preprint arXiv:1909.03291* (2019).

[17] Anshuman Dutt and Jayant R Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1039–1050.

[18] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Foundations and Trends in Databases* 8, 1-2 (2017), 1–130. https://doi.org/10.1561/1900000058

[19] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.* 40, 4 (Jan. 2012), 45–51. https://doi.org/10.1145/2094114.2094126

[20] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.* 4, 2 (Nov. 2010), 105–116. http://dl.acm.org/citation.cfm?id=1921071.1921077

[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. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1917–1923. https://doi.org/10.1145/2723372.2742795

[22] Jayant R. Haritsa. 2010. The Picasso Database Query Optimizer Visualizer. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 1517–1520. https://doi.org/10.14778/1920841.1921027

[23] Joseph M. Hellerstein. 1998. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems* 23, 2 (June 1998), 113–157.

[24] Denis Hirn and Torsten Grust. 2020. PL/SQL Without the PL. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2677–2680.

[25] InfluxData Inc. 2019. Time series database (TSDB) explained. https://www.influxdata.com/time-series-database.

[26] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. 2007. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *PACT*. 189–198.

[27] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *PVLDB* 8, 6 (Feb. 2015), 642–653.

[28] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. ACM, New York, NY, USA, 106–117. https://doi.org/10.1145/276304.276315

[29] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499. https://doi.org/10.1145/1454159.1454211

[30] Michael Kalloniatis and Charles Luu. [n.d.]. Temporal Resolution. https://webvision.med.utah.edu/book/part-viii-psychophysics-of-vision/temporal-resolution/.

[31] Srinivas Karthik, Jayant R Haritsa, Sreyash Kenkre, and Vinayaka Pandit. 2018. A concave path to low-overhead robust query processing. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2183–2195.

[32] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[33] 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, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[34] Changkyu Kim et al. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. *PVLDB* 2, 2 (Aug. 2009), 1378–1389.

[35] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-level Language. *Proc. VLDB Endow.* 7, 10 (June 2014), 853–864. https://doi.org/10.14778/2732951.2732959

[36] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 197–208.

[37] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. 613–624. https://doi.org/10.1109/ICDE.2010.5447892

[38] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. 613–624.

[39] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13.

[40] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 311–326. https://doi.org/10.1145/2882903.2882925

[41] Per-Åke Larson, Mike Zwilling, and Kevin Farlee. 2013. The Hekaton Memory-Optimized OLTP Engine. *IEEE Data Eng. Bull.* 36, 2 (2013), 34–40. http://sites.computer.org/debull/A13june/Hekaton1.pdf

[42] Stefan Manegold, Peter Boncz, and Martin Kersten. 2000. Optimizing database architecture for the new bottleneck: memory access. *J. VLDB* 9, 3 (2000), 231–246.

[43] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *TKDE* 14, 4 (July 2002), 709–730.

[44] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (June 2011), 539–550.

[45] Oracle Corp. 2019. GraalVM. https://www.graalvm.org/.

[46] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java hotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Monterey, California) *(JVM'01)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1267847.1267848

[47] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *PVLDB* 11, 6 (2018), 663–676.

[48] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR 2017, Conference on Innovative Data Systems Research*. http://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf

[49] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*. 1493–1508.

[50] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *SIGMOD*. 755–766.

[51] Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom Filters for Advanced SIMD Processors. In *DaMoN*. Article 6.

[52] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *DaMoN*. Article 9.

[53] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[54] Kenneth A. Ross. 2004. Selection Conditions in Main Memory. *ACM Transactions on Database Systems* 29, 1 (2004), 132–161.

[55] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 1231–1242. https://doi.org/10.1145/2463676.2465292

[56] Nadathur Satish et al. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*. 351–362.

[57] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2020. Dynamic speculative optimizations for SQL compilation in Apache Spark. *Proceedings of the VLDB Endowment* 13, 5 (2020), 754–767.

[58] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*. 1961–1976.

[59] Doug Simon. [n.d.]. libgraal: GraalVM compiler as a precompiled GraalVM native image. https://medium.com/graalvm/libgraal-graalvm-compiler-as-a-precompiled-graalvm-native-image-26e354bee5c.

[60] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The Architecture of SciDB. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management* (Portland, OR) *(SSDBM'11)*. Springer-Verlag, Berlin, Heidelberg, 1–16. http://dl.acm.org/citation.cfm?id=2032397.2032399

[61] Tableau Inc. 2019. Tableau. https://www.tableau.com.

[62] Sandeep Tata. 2007. *Declarative Querying for Biological Sequences*. Ph.D. Dissertation. Ann Arbor, MI, USA. Advisor(s) Patel, Jignesh M. AAI3276308.

[63] Thomas Willhalm et al. 2009. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB* 2, 1 (Aug. 2009), 385–394.

[64] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. 13–14.

[65] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 187–204.

[66] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. 2016. Non-invasive Progressive Optimization for In-memory Databases. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1659–1670. https://doi.org/10.14778/3007328.3007332

[67] Wangda Zhang and Kenneth A Ross. 2020. Exploiting data skew for improved query performance. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[68] Wangda Zhang and Kenneth A Ross. 2020. Permutation Index: Exploiting Data Skew for Improved Query Performance. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1982–1985.

[69] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of SIGMOD Conference*.

[70] M. Zukowski, M. van de Wiel, and P. Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. 1349–1350. https://doi.org/10.1109/ICDE.2012.148