# Hazard-Adaptive Query Processing Using Hardware-Assisted Runtime Profiling

## ABSTRACT

Operator performance in in-memory Data Management Systems often suffers from micro-architectural hazards such as cache misses and branch mispredictions. While many operators have alternative implementations that are robust against such hazards, these generally perform worse when no hazards are encountered. Unfortunately, hazards are caused by data characteristics that query optimizers cannot easily capture (most importantly auto-correlation) making static hazard-conscious optimization difficult. In addition, statically optimized plans suffer from overhead when data characteristics vary within a table.

To address both of these problems, we propose a hazard-adaptive approach to query execution. Through hardware-assisted runtime profiling of low-level metrics operators can dynamically adapt to "hazardous" data. We propose a conceptual framework for hazard-adaptive operators and implement our approach in a prototypical relational data processor. We demonstrate that hazard-adaptive operators provide robustness to a range of data characteristics by identifying the optimal static implementation and even adapting to changing patterns within the data.

**ACM Reference Format:**
. 2023. Hazard-Adaptive Query Processing Using Hardware-Assisted Runtime Profiling. In *Proceedings of 2024 ACM SIGMOD International Conference on Management of Data (SIGMOD '24).* ACM, New York, NY, USA, 13 pages.

## 1 INTRODUCTION

Performance of data analytics is determined by two factors: queries and data characteristics. As the number of operators supported by a system is usually moderate, capturing query characteristics of a dataflow execution engine is comparatively easy. Accurately formalizing data characteristics, however, is significantly harder.

Formalizations of data characteristics are usually tailored to a specific use: logical query optimization, e.g., the optimization of relational operators that are, by definition, order-invariant. Consequently, the data characterization does not require the capturing of order either. Simple histograms (usually built on sampled values) suffice to estimate single-attribute cardinalities required for join- or selection-order optimization. To accurately capture complications like correlation, histograms need to be multidimensional and samples larger, but the principle remains the same.

However, many aspects of query processing are only moderately impacted by selectivity and attribute correlation but are highly affected by *auto-correlation*, i.e., patterns in the data. In particular, the selection of an appropriate (physical) algorithm to implement a logical operator is very sensitive to patterns in the data. To illustrate this, consider Figure 1 which illustrates the effect of the sortedness of input data on the performance of two well-known implementations of the predicated selection operator: one branching and one branch-free (both described by Ross [29]). The experiment shows that, at equal selectivity, the sortedness of the data can have a roughly 10x performance impact on the operator. We found that many other operators suffer from similar problems.

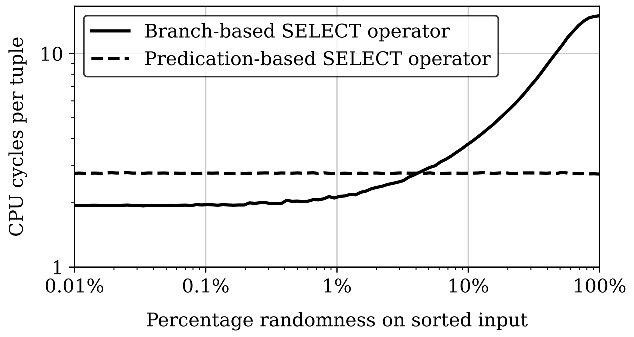


**Figure 1: Performance of SELECT at a fixed 50% selectivity for 250M 32-bit integers on a conventional server**

Capturing auto-correlation such as sortedness, clusteredness, or, generally, patterns in the data is more involved. As histograms are order-invariant, they do not capture auto-correlation, and extending them to do so is far from trivial [25]. While other approaches to measuring the sortedness of data exist, these tend to be complicated and brittle [1, 5, 14]. Any approach that builds on these techniques would inherit these traits.

An alternative approach to selecting an appropriate algorithm could be based on the idea of micro-adaptivity [30]. Under that paradigm, the operator implementation could be selected on a batch-by-batch basis: the system would observe the performance of the operator and use an exploration/exploitation-based scheme to select the best-performing implementation. State-of-the-art micro-adaptive processing engines treat operators as black boxes and use execution time to capture utility/reward. While this approach has proven effective in selecting the best-performing binary for the same source code (generated using different compilers) [30], it has substantial shortcomings. First, it is limited to selecting one of multiple interchangeable implementations, leaving a large part of the solution space unexplored. Second, the approach is complicated and brittle (leaving the operator decision to a learned model solving the multi-armed bandit problem). Third, it takes many iterations (i.e., processed batches) to determine the optimal implementation – if patterns in the data change frequently, the approach would "flip-flop" between suboptimal implementations.

To address these problems, we propose "hazard-adaptive" operators which are treated as "white" boxes in a system that continuously monitors them and "advises" when a switch to a more hazard-robust implementation is prudent. To this end, we make the following technical contributions:

- We present a conceptual operator framework for developing hazard-adaptive operators. We describe the principles underpinning our runtime framework and define a generalized operator interface and a unified interface for all foundational algorithms.
- Based on the insight that micro-architectural hazards are a strong predictor for overall performance, we propose to monitor performance counters at runtime and micro-adaptively select the most appropriate processing algorithm.
- We apply our framework to develop hazard-adaptive operators for the three most prominent relational operators (`SELECT`, `GROUP BY`, and `JOIN`). We evaluate their performance and demonstrate their robustness across a broad range of input data characteristics.

The paper is organized as follows: In Section 2, we establish the necessary background knowledge. Section 3 presents our hazard-adaptive operator framework, including its architecture and interfaces. We develop three hazard-adaptive operators using this framework in Section 4 and discuss implementation details in Section 5. We evaluate the performance of our hazard-adaptive operators in Section 6. Section 7 highlights distinctions from existing approaches, and we conclude in Section 8.

## 2 BACKGROUND

Before introducing our hazard-adaptive operator framework we provide essential background on micro-architectural hazards, micro-architecturally efficient algorithms, and performance counters.

### 2.1 Micro-Architectural Hazards

Superscalar execution enhances performance by executing multiple CPU instructions concurrently. This relies on high pipeline utilization to minimize stall cycles.

Micro-architectural hazards delay or interrupt the execution of instructions. To develop high-performing, micro-architecturally efficient algorithms, it is crucial to minimize these hazards. The three main types are:

**(1) Control hazards** are caused by a change in control flow, such as conditional branches or jumps. This can lead to stall cycles due to bad speculation and the subsequent pipeline flush, or due to a lack of eligible instructions associated with control-flow dependencies.

**(2) Data hazards** occur when the instruction pipeline is stalled due to operands (data) not being ready at the time of execution. Such stalls result from cache or TLB misses since the retrieval of data from main memory is slower than the speed at which a CPU executes instructions.

**(3) Structural hazards** stem from limited CPU resources which results in multiple instructions requiring access to the same hardware resource simultaneously. For example, an ALU stall can occur when executing resource-intensive instructions.

### 2.2 Micro-Architecturally Efficient Algorithms

Micro-architecturally efficient algorithms maximize CPU efficiency by minimizing hazards. The principles underpinning these algorithms fall into four categories:

**(1) Data Locality and Cache Awareness:** Maximizing cache utilization minimizes data hazards and prevents data stalls. This can be achieved by increasing data locality to prevent unnecessary cache misses, increasing cache line utilization, reducing the overall cache footprint of the program, and prefetching data where possible.

**(2) Minimizing Control Flow and Branches:** Control hazards through bad speculation can be removed by writing branch-free code. Limiting the use of `JUMP` instructions minimizes frontend stalls due to a lack of eligible instructions.

**(3) Parallelism and Vectorization:** Instruction-level parallelism can be maximized by reordering instructions to minimize dependencies and therefore maximize the utilization of execution units. The use of vectorization to execute instructions statically parallel by performing operations on multiple data elements in parallel further increases utilization of all pipeline stages, reduces instruction overhead, and improves memory bandwidth utilization.

**(4) Optimizing Memory Access Patterns:** A high cache miss rate is typically due to a random memory access pattern. Switching to a sequential access pattern greatly reduces the cache miss rate.

Performance engineering techniques apply these principles to increase an algorithm's robustness to hazards and thereby develop micro-architecturally efficient algorithms. However, this is typically associated with a trade-off of decreased algorithmic efficiency.

### 2.3 Hardware Performance Counters

Hardware performance counters (performance counters) are a set of registers located within the performance monitoring unit (PMU) of modern CPUs. These counters can be configured to count specific micro-architectural events during program execution, enabling the profiling of software at runtime [21, 33, 35].

The key advantage of performance counters lies in their low execution cost. Malone et al. confirm that the runtime performance overhead is virtually zero [21]. This is corroborated by Zeuch et al. [36] and in the official Intel documentation [10].

According to Eranian [13], performance counters are the crucial resource for identifying and solving performance issues, especially those related to the memory subsystem. Zeuch et al. use this approach, making use of the non-invasive monitoring provided by performance counters, to implement progressive optimization, dynamically adjusting query plan execution based on runtime characteristics and feedback [23].

## 3 HAZARD-ADAPTIVE OPERATOR FRAMEWORK

The optimal operator implementation is dependent on the characteristics of the input data and the consequent frequency of hazards that it generates. Our objective is to design an operator that can switch between alternative, output equivalent, implementations at runtime such that it is micro-adaptive in response to hazards.

In the context of our work, we distinguish two categories of implementation, "hazard-affected" implementations and "hazard-robust" implementations.

Hazard-adaptive operators are formed by combining algorithms from these two categories. We subsequently refer to these algorithms as the "foundational algorithms" of that hazard-adaptive operator. In this section, we introduce these algorithmic categories and present a conceptual operator framework for how a hazard-adaptive operator can accurately and efficiently switch between the foundational algorithms at runtime.

## 3.1 Hazard-Affected Algorithms

The defining principle of hazard-affected algorithms is the minimization of work i.e., CPU instructions executed. This is achieved by focusing on the efficiency of the algorithmic approach.

As a result, the performance of this category of algorithm is dependent on the input data characteristics for two reasons. Firstly, work is only executed if the data necessitates it, therefore the number of instructions executed depends on the data. Secondly, this frequently introduces control-flow dependencies (to determine whether or not to execute work) which makes the algorithm susceptible to control hazards. Moreover, algorithms within this category almost universally do not focus on micro-architectural efficiency, instead, they focus on work efficiency. Consequently, they are susceptible to all of the micro-architectural hazards described in Section 2.1. The generation of these hazards is also determined by the input data characteristics. Overall, this dependency of performance on the input data characteristics means that the performance of hazard-affected algorithms is highly variable.

For example, a branching implementation of the SELECT operator has highly variable performance for both of these reasons. Firstly, the quantity of work is determined by the selectivity of the input, with the output being generated only if the selection predicate is satisfied. Secondly, the performance at a given selectivity is determined by the sortedness of the input due to the presence of control-flow dependencies and the resulting control hazards. The performance impact of this is illustrated in Figure 1. This variability in performance means that, for certain input data characteristics, an alternative implementation is superior.

## 3.2 Hazard-Robust Algorithms

The alternative category of algorithms are hazard-robust algorithms. Unlike hazard-affected algorithms, the key principle of hazard-robust algorithms is micro-architectural efficiency and robustness.

The performance of hazard-robust algorithms is far less dependent on the input data compared to hazard-affected algorithms. Their robustness to hazards ensures consistently high CPU instruction pipeline throughput i.e., instructions are executed at a rate that is largely unaffected by the input data characteristics. Additionally, as we will illustrate in Section 6, hazard-robust algorithms are typically designed such that the quantity of work that they perform is largely independent of input data characteristics. This is demonstrated by the predication-based implementation of the SELECT operator in Figure 1 which has performance that is independent of the sortedness of the input data.

There exist a number of techniques [6, 11, 12, 26, 28, 29, 32] which implement the principles of micro-architecturally efficient algorithms as described in Section 2.2, and can be used to develop hazard-robust versions of hazard-affected algorithms.

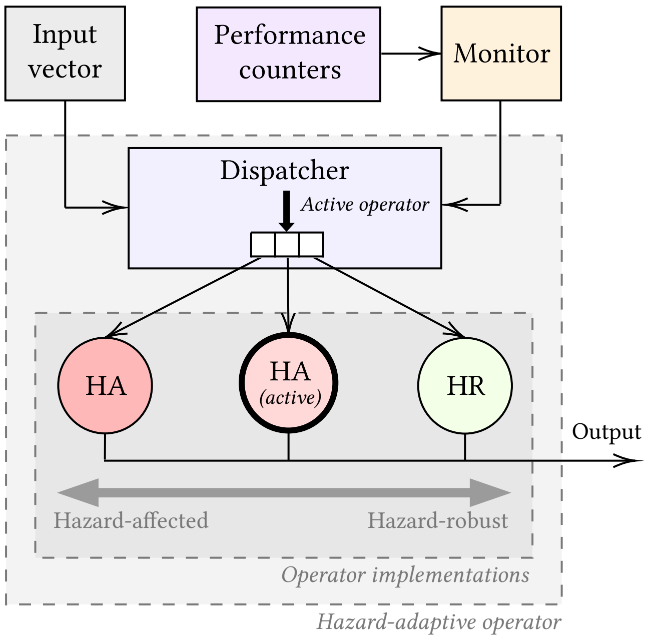


**Figure 2: Hazard-adaptive operator architecture schematic**

## 3.3 Hazard-Affected Operator Architecture

The architectural schematic in Figure 2 illustrates how we combine algorithms from these two categories into a hazard-affected operator. The foundational algorithms require a single hazard-robust (HR) algorithm and at least one hazard-affected (HA) algorithm.

The "dispatcher" contains an array of these foundational algorithms with an offset to identify the currently active operator implementation. The dispatcher is responsible for selecting the operator implementation to process micro-batches of the input vector.

As we describe in the following sections, the dispatcher is directed by the "monitor" that reads the performance counters and, based on a set of heuristics, sends a signal to the dispatcher to adjust the hazard-robustness of the hazard-adaptive operator if necessary.

## 3.4 Increasing Robustness

The initial implementation for the hazard-adaptive operator is the most hazard-affected algorithm. This is because, as shown in this section, the monitor can use the generation of hazards to determine when it is preferable to switch to a more hazard-robust algorithm.

*Determining the Dominant Micro-Architectural Event.* The frequency of micro-architectural hazards is highly correlated with the performance of a hazard-affected algorithm: hazards lead to reduced instruction pipeline utilization and therefore directly increase the time taken to execute a set of instructions.

We define the dominant micro-architectural event as the event that is primarily responsible for generating the hazards that degrade the algorithm's performance.

As introduced in Section 2.3, performance counters can be used to monitor the frequency of this event whilst incurring minimal overhead. By periodically calculating the per-tuple frequency of the dominant event, the monitor can assess the current performance of the hazard-affected algorithm.

A key advantage of such monitoring of operator performance is that it monitors the key data characteristic that drives performance rather than a higher-level characteristic which only gives a partial view of performance. For example, as shown in Figure 1 for SELECT, sortedness (or more precisely predictability) is the key data characteristic rather than simply selectivity. Measuring the dominant event using performance counters directly captures these order-dependent patterns in the data, rather than higher-level order-invariant metrics.

*Determining Tipping Points.* Given the ability to evaluate the current runtime performance of a hazard-affected algorithm, a heuristic is required to identify the threshold event frequency at which a switch to a more hazard-robust algorithm is superior.

The specific heuristic required depends on whether one or two tipping points (crossovers) are present in the performance space of the operator implementations.

The location of tipping points is influenced by a number of factors: the underlying hardware which determines the cost of hazards, the input data type size which alters the relative trade-offs between different implementations, and the degrees of parallelism due to the shared usage of finite resources.

Each combination of these factors requires a unique heuristic, however, these heuristics are constant and do not require dynamic reconfiguration. They can be considered machine constants. This is due to most resources being allocated on a per-core basis and therefore being isolated from other processes. The primary shared resources are the shared cache and memory bandwidth.

Where hazard-affected algorithms have the potential to suffer a high cache miss rate due to shared cache usage then last-level cache misses will be the dominant event and the hazard-adaptive operator will be naturally adaptive to the level of contention of the shared cache (observed in Figure 6c, Section 6.3). We do not consider memory bandwidth to be a resource on which heuristics should be dynamically reconfigured. Firstly, operators become memory-bound at high DOP, however, this is already accounted for since a unique heuristic is required for each DOP (observed in Figure 6b, Section 6.3). Secondly, memory-bound impacts due to other processes are likely to be highly variable, and therefore dynamically reconfiguring a heuristic to be used for the entire operator runtime risks driving further sub-optimal adaptive behavior.

The heuristics for operators can be determined with a calibration function that runs prior to the first workload on a new machine.

## 3.5 Dispatching to Foundational Algorithms

Using these heuristics combined with performance counter readings, the monitor provides information to the hazard-adaptive operator using the interface defined in Listing 1.

The dispatcher combines this information with additional switching logic to select the operator implementation to process the next micro-batch via the push-based foundational algorithm interface in Listing 2. The dispatcher also ensures that progress is efficiently maintained across switches.

```
enum Event {BranchMispredictions, LastLevelCacheMisses,
            L2TlbStoreMisses, ...};

template <class InputType, class OutputType>
class HazardAdaptiveOperator {
    HazardAdaptiveOperator(InputType *input,
                           size_t maxOutputSize);
    size_t processMicroBatch(size_t n);
    void adjustRobustness(Event event, int adjustment);
    vector<OutputType> getOutput();
};
```

**Listing 1: Hazard-adaptive operator interface**

```
template <class InputType, class OutputType, class State>
class OperatorImplementation {
    OperatorImplementation(size_t inputSize,
                           size_t maxOutputSize,
                           State &&state = {});
    size_t processMicroBatch(InputType *input, size_t n);
    vector<OutputType> getOutput();
    State getState() && { return std::move(state); }
    State state;
};
```

**Listing 2: Foundational algorithm interface**

*Hazard-Adaptive Operator Interface.* The monitor communicates with the hazard-adaptive operator via the interface in Listing 1. The monitor is able to adjust the robustness of the hazard-adaptive operator in response to changes in the frequency of micro-architectural events. This separates the heuristics controlling hazard-adaptivity from the internal switching logic of the hazard-adaptive operator.

A hazard-adaptive operator may use multiple hazard-affected algorithms. For example, multiple discrete implementations or individual points along a spectrum of hazard-affected implementations (e.g., the radix partition algorithm introduced in Section 4.3).

The switching logic between algorithms is encapsulated within the dispatcher which responds to requests to adjust the robustness to a specific event by selecting the operator implementation object to send subsequent micro-batch processing requests to. For example, for a spectrum of hazard-affected algorithms, when the robustness is increased a switch can be made to the next best-performing hazard-affected algorithm and eventually the hazard-robust algorithm.

The monitor adjusts the robustness of the operator by specifying a particular event and passing in an integer to indicate the magnitude and direction of the adjustment required. Although not implemented in this paper, this means that foundational algorithms can be associated with multiple hazards, expanding the scope of possible foundational algorithms that can be included.

These foundational algorithms are encapsulated within the hazard-adaptive operator object which is responsible for their construction and initialization.

*Maintaining Progress When Switching.* At some point, the results from the individual foundational algorithms must be merged. Since a switch was made to the hazard-robust algorithm (hence the required merge), at least some of the data is not suitable for processing using a hazard-affected algorithm. Therefore, the hazard-robust algorithm is used to merge the results of the foundational algorithms.

If results are merged immediately when a switch is made, work is duplicated by repeatedly merging the same results every time a switch occurs. To prevent this, merges are deferred until after the entire input has been processed. A single merge is then performed by the hazard-robust algorithm. Note that it is typically preferable to defer the processing of micro-batches by the hazard-robust algorithm, and instead include the processing of these micro-batches as part of the merge phase. However, this is operator-specific.

The dispatcher executes merges by accessing the internal state of hazard-affected operator implementations (which contain the collated output) and moving these to the hazard-robust operator using the foundational algorithm interface in Listing 2.

*Foundational Algorithm Interface.* An efficient hazard-adaptive operator requires the reuse of intermediate results created by operator implementations across the processed micro-batches. For example, where memory is allocated (e.g., a hash table) it should be reused for all processing by that operator implementation. To achieve this we encapsulate each foundational algorithm into an object which maintains these resources between processing different micro-batches. The interface for these objects in Listing 2 defines a unified interface for the dispatcher to interact with.

At the point of construction of the hazard-affected algorithm objects, the dispatcher does not know how many tuples will be processed with that specific object. Therefore, the initialization cost should be $O(1)$ complexity. This means that any memory allocations must be lazily initialized to ensure that initialization costs are only incurred if the memory is used.

Alternatively, when processing with the hazard-robust algorithm is deferred until after the hazard-affected algorithm pass is complete, the dispatcher knows the number of tuples that will be processed by the hazard-robust algorithm. Therefore, the initialization cost can be $O(n)$ with respect to the number of input tuples.

Each foundational algorithm object has an internal state where the operator collates the results from each `processMicroBatch()` call. The dispatcher uses the `getState()` function to extract the internal state from one operator implementation and transfer it to another. Note that the `getState()` function is "r-value reference qualified" (indicated by the &&-suffix) which, effectively, encodes that that function can only be invoked on objects that are "marked for destruction" (using a `std::move`). This ensures (at compile-time) that, once the state is extracted, the operator can no longer process data (the required semantics). When a new operator is instantiated, it optionally takes a `State&&`-parameter. The r-value qualification implies that the operator implementation takes ownership of an existing object. This is required to efficiently (i.e., copy-free) pass the state to a newly instantiated operator implementation. When such a state-migration takes place, processing using the hazard-robust algorithm is deferred, and thus the necessary states from the other operator implementations can be passed in via the constructor.

## 3.6 Decreasing Robustness

As well as increasing the robustness of a hazard-adaptive operator, it should be possible to decrease the robustness of the operator. This is because, despite having previously required increased robustness, the input might have a distribution that merits switching back to more hazard-affected algorithms and increasing work efficiency at the expense of micro-architectural efficiency.

If a hazard-affected algorithm is being executed and generating few hazards the monitor can signal to the dispatcher that it would be beneficial to switch to a more hazard-affected algorithm by using a negative adjustment value for the `adjustRobustness()` call.

However, if the hazard-robust algorithm is being executed the design principle of micro-architectural efficiency means that the work executed is entirely independent of the key data characteristic. This is illustrated in Figure 1 where the performance of the predication-based implementation is independent of the predictability of the data. As a result, the performance counters provide no useful information to the monitor and the dispatcher cannot make an informed decision of when to switch to an alternative algorithm. This leaves two methods for switching from hazard-robust algorithms:

**(1)** In some cases where higher-level order-invariant metrics (e.g., selectivity) can be determined at runtime these can be used to infer key data characteristics and therefore identify a subset of cases where it is preferable to switch back to the hazard-affected algorithm (e.g., 0% selectivity implies full predictability).

**(2)** The dispatcher can choose to periodically perform probes (tests) using a hazard-affected algorithm which will switch optimally. These probes should include the minimum number of tuples necessary to obtain a statistically reliable value for the dominant event frequency.

Note that it is these periodic returns to a hazard-affected algorithm that means that the number of tuples to be processed by the hazard-robust algorithm is known without actually processing the tuples. It is this knowledge that allows the dispatcher to defer processing by the hazard-robust algorithm until after processing by the hazard-affected algorithms is complete.

## 4 HAZARD-ADAPTIVE OPERATORS

Creating hazard-adaptive operators is non-trivial, requiring the selection of appropriate foundational algorithms, identification of the dominant micro-architectural event(s), development of accurate heuristics for the monitor, and efficient implementation switching by the dispatcher. In this section, we develop three such operators, discussing their implementation within our framework.

The problem space for hazard-adaptive operators is broad, encompassing various operators and implementations that are specific to each database management system. To validate our concept of hazard-adaptive operators, we cover the three most impactful relational operators, `SELECT`, `GROUP BY`, and `JOIN`, collectively covering a substantial portion of this space.

### 4.1 SELECT operator

*Foundational Algorithms.* A hazard-affected implementation of the `SELECT` operator is the standard branching algorithm defined as: `if(/*condition*/) output[j++] = input[i++];`. Work is minimized by only updating the output if the conditional predicate is satisfied. However, the use of a conditional can result in branch mispredictions (control hazards) leading to pipeline flushes [24].

To create an alternative branch-free implementation the technique of predication can be applied, always copying to the output: `output[j] = input[i++]; j += (/*condition*/);`. This makes the algorithm robust to control hazards at the cost of the extra work of updating the output before testing the conditional predicate.

Predication transforms control hazards into data hazards [29]. Therefore, despite being substantially less susceptible to hazards, they can be further reduced by applying vectorization [38].

The SELECT operator can either return the indexes of keys that satisfy the conditional predicate (SELECT_Indexes) or the corresponding payload (SELECT_Values). SELECT_Values is at higher risk of becoming memory-bound given that a payload is required, whereas SELECT_Indexes only requires reading of the keys. Therefore, we create two versions of the SELECT operator to reflect these different approaches. We use just predication for the hazard-robust implementation of SELECT_Indexes and both predication and SIMD for the hazard-robust implementation of SELECT_Values.

For both branch-free algorithms, the number of instructions per tuple is constant and their performance is only dependent on the selectivity of the data which determines the size of the output and consequently, additional reads and writes.

*Monitoring Heuristics.* The dominant micro-architectural event for the branching algorithm is branch mispredictions which occur when the branch predictor does not work effectively. Branch mispredictions dominate the performance since the resulting pipeline flushes cause costly pipeline bubbles in the instruction pipeline.

Branch mispredictions correlate with the (order-dependent) key data characteristic of predictability, rather than the higher-level (order-invariant) metric of selectivity. Figure 1 illustrates the importance of distinguishing these two metrics as varying the predictability of the input data at a fixed selectivity impacts the performance of the branching implementation due to branch mispredictions.

The monitoring heuristic is defined by a threshold value for the frequency of branch mispredictions. When this threshold is breached the monitor sends a signal to the dispatcher to increase the operator robustness i.e., switch to the branch-free algorithm.

Only a single tipping point is present for SELECT_Values where at high selectivity values the vectorization-based algorithm outperforms the branching implementation. The heuristic for this is straightforward, with a single threshold value being sufficient.

For SELECT_Indexes, we observe two tipping points, one towards each extreme of selectivity. In this case, a heuristic is required that links together the branch misprediction frequency thresholds at these two crossover points such that the heuristic is effective at all selectivity values. The monitor uses linear interpolation to calculate the branch misprediction frequency between these points at runtime based on the current selectivity. This interpolation accounts for the linear variation in the performance of the branch-free algorithm with selectivity.

*Dispatching Logic.* The dispatcher switches to processing micro-batches with the branch-free algorithm when it receives a signal from the monitor.

The dispatcher also contains logic for switching back to the branching algorithm based purely on the (order-invariant metric) selectivity. For example for SELECT_Values, above a selectivity of ∼80% the hazard-robust algorithm is always superior to the hazard-affected algorithm, even for fully predictable inputs.

After processing a set number of tuples with the branch-free algorithm, the dispatcher runs a probing phase by switching back to the branching algorithm for a small micro-batch to determine whether the input has changed sufficiently for the branching algorithm to now be superior.

## 4.2 GROUP BY operator

*Foundational Algorithms.* A hazard-affected implementation of the GROUP BY operator is a hash-based grouping algorithm. Tuples are inserted into a hash table[1] and work is minimized by aggregating a payload immediately with the corresponding existing aggregated payload in the hash table. An aggregation function is passed in as an argument, and the algorithm returns an unordered array of unique keys with their corresponding aggregated payload.

This algorithm is susceptible to data hazards in the form of capacity cache misses leading to data stalls. Capacity cache misses occur due to a random memory access pattern on a data structure that exceeds the size of the cache (i.e., the working set cannot fit into the cache). This leads to "cache thrashing" due to the frequent swapping of data into and out of the cache.

To create an alternative hazard-robust algorithm the random memory access pattern must be replaced with a sequential access pattern. This can be achieved by using a multi-pass algorithm which will allow for prefetching data and improve cache line utilization.

Radix sort is a multi-pass algorithm that can be used to create a sort-based implementation. Once the input is sorted a final pass is performed and adjacent tuples with matching keys are aggregated. This sort-based algorithm reduces the cache footprint of the operator, thus preventing data stalls. However, this comes at the cost of unnecessarily sorting the output.

Care must be taken to implement an efficient and hazard-robust sort-based algorithm. For example, we combine the aggregation pass with the final radix sort pass saving two sequential passes: the final data shuffling pass and the separate aggregation pass. Additionally, the fan-out, defined as $2^{radixBits}$, which determines the number of partitions (and therefore pages) written to during the data shuffling phase of radix sort, must be below that of the sTLB[2] capacity to not risk sTLB store misses (a data hazard).

The performance of the sort-based hazard-robust algorithm is dependent on two factors: Firstly, the size of the output which is defined by the (order-invariant) metric of cardinality. This is similar to SELECT where the performance of the hazard-robust algorithms is dependent on the size of the output, defined by the selectivity of the workload. Secondly, radix sort has a step-wise performance profile depending on the number of passes required, determined by the number of bits required to sort the input data[3]. In theory, the heuristic should be modified at runtime to account for changes in the number of passes required compared to the number used in the calibration of the heuristic. However, the tipping point is likely to always occur at the same number of passes and therefore we found that a simple fixed value closely followed the optimal static implementation. We discuss this further in Section 5.3.

*Monitoring Heuristics.* The dominant micro-architectural event for the hash-based hazard-affected algorithm is last-level cache misses

[1]We assume that the cardinality of the input is known a priori and the hash table is initialized such that no resizing operations are required.
[2]Shared TLB, also known as unified TLB (uTLB), or L2 TLB.
[3]We include a single pass to identify the number of bits required to sort the input data to prevent any unnecessary radix sort passes.

since of all the cache levels, it is the pipeline stalls due to last-level cache misses that dominate the performance of the algorithm.

Last-level cache misses correlate with the (order-dependent) key data characteristic of variability, rather than the higher-level (order-invariant) metric of cardinality. The difference between these two metrics can be illustrated by inputs of equal cardinality but with different levels of clustering[4]. The clustered input has lower variability, i.e., on average a smaller difference between adjacent keys. Therefore, adjacent keys will be hashed to buckets that are relatively close to each other and consequently, are more likely to be already resident in the cache thus reducing cache misses. This phenomenon occurs due to locality-preserving hashing [18] which is generally a desired property of hash-based data structures since it can improve cache efficiency.

The extent of this effect depends on the hash function used. In our implementation, we used Tessil's `robin_map` [31] which outperformed other hash tables[5] across a range of cardinality values and input data types. This hash map includes a degree of locality preservation and utilizes "Robin Hood" hashing [8].

For `GROUP BY`, tipping points can only be identified using the last-level cache miss rate measured by performance counters. This is because the higher-level metric of cardinality (which would be able to identify a subset of tipping points) is only determined after processing is fully complete. This is unavoidable, as to fully process a tuple requires processing all other tuples as well. This is in contrast to `SELECT` where tuples can be processed independently.

Since only a single tipping point is present for `GROUP BY`, the monitor only requires a simple heuristic that sends a signal to the dispatcher when the last-level cache miss rate breaches a predetermined threshold.

*Dispatching Logic.* Considering the foundational algorithm interface described in Section 3.5, the initialization of the hash table for the hazard-affected algorithm should be $O(1)$ complexity. However, the `robin_map` used in our implementation is eagerly initialized meaning that the $O(outputSize)$ memory allocation and initialization occurs at the point of construction. We leave the modification of this hash map (which could reduce the overhead when the hazard-robust algorithm is required) for future work.

The hash map is reused for all micro-batches that are processed by the hazard-affected algorithm, and the hazard-robust algorithm is deferred until after the hazard-affected algorithm has processed all the necessary micro-batches. The sort-based algorithm then processes the remaining micro-batches of the input in addition to the results collated in the hash map which forms the state of the hash-based algorithm. This state is passed into the hazard-robust algorithm at construction using move semantics as described in Section 3.5. At construction, the additional buffer required by radix sort can be eagerly initialized since the dispatcher knows the total number of tuples that will be processed.

When executing the hazard-robust algorithm, the dispatcher uses the strategy of periodically running a probing phase, switching back to the hazard-affected algorithm as described in Section 4.1.

[4]Defined as the similarity between adjacent values e.g., clustering of 1,000 means that neighboring values would be $\leq$ 1,000 positions apart if the input array were sorted.
[5]Google `dense_hash_map`, Abseil `flat_hash_map`, Tessil `robin_map`, Tessil `hopscotch_map`, Facebook `F14_hash_map`.

## 4.3 `JOIN` operator

Given the breadth of implementations of the `JOIN` operator, we constrained our investigation of the application of hazard-adaptivity to the `PARTITION` phase of the radix hash join algorithm [22]. As we will show, `PARTITION` is a point in the problem space that is non-trivial and, we believe, is highly relevant to other operators across the problem space [37].

*Foundational Algorithms.* The work required for radix partitioning can be minimized by using a large fan-out, defined as $2^{radixBits}$, such that the data can be partitioned with a single partitioning pass. However, a large fan-out is susceptible to data hazards in the form of sTLB store misses during the data shuffling phase of partitioning.

During the data shuffling phase values are moved to their respective partition. Each partition typically resides on its own page, therefore requiring a TLB entry [3]. If the fan-out exceeds the sTLB capacity and a random partition access pattern is required then sTLB store misses occur causing data stalls. A high sTLB store miss rate is known as "TLB thrashing" [4] (analogous to cache thrashing).

Similarly to how the use of a multi-pass algorithm reduced the cache footprint for `GROUP BY`, the TLB footprint of the hazard-affected `PARTITION` algorithm can be reduced by converting it to a multi-pass algorithm simply by reducing the number of radix bits. By ensuring that the TLB footprint (fan-out) is smaller than the sTLB capacity a hazard-robust version of the previously hazard-affected algorithm can be created.

By making the algorithm robust to hazards we ensure that the performance per tuple is broadly independent of the input data. However, unlike for `SELECT` and `GROUP BY`, there is a spectrum of hazard-affected algorithms, each with a different radix bits value between that of the fully hazard-robust algorithm and that of the single pass algorithm. By taking advantage of this spectrum of algorithms we can maximize the range of inputs that the hazard-adaptive operator can process optimally. We therefore create a separate hazard-affected algorithm for each of these radix bit values.

*Monitoring Heuristics.* The dominant micro-architectural event for high fan-out radix partitioning is sTLB store misses. These are last-level TLB store misses since these have the greatest performance penalty, requiring main memory access to query the page table.

The sTLB store miss rate correlates with the (order-dependent) key data characteristic of "partitionedness" with respect to the current radix bits value. In this context the term partitionedness[6] indicates how close the data is to already being partitioned and encompasses a number of metrics. These metrics influence the sTLB store miss rate and include metrics such as clusteredness and sortedness, or order-invariant metrics such as skew and cardinality.

The monitor adjusts the hazard-adaptive operator's robustness when TLB thrashing is observed via the sTLB store miss rate.

*Dispatching Logic.* The optimal number of radix bits to partition an input depends on the pre-existing partitionedness of the data, with greater existing partitionedness meaning that a larger number of radix bits can be used without generating excessive data hazards.

[6]Note that we use clusteredness in place of partitionedness for `PARTITION` related plots given its tighter definition and greater real-world applicability.

As for all hazard-adaptive operators, the dispatcher starts by executing micro-batches on the most hazard-affected algorithm i.e., the operator implementation with the highest radix bits value.

When the dispatcher receives a signal to increase robustness it switches the active operator implementation to an algorithm with a lower number of radix bits. At this point the necessary partitions are merged to match the partitions associated with the new radix bits value and the associated histogram values are updated.

Note that this simplified approach limits the operator's use to homogeneous input data[7]. We leave the extension to full adaptivity (required for the processing of heterogeneous data) for future work.

## 5 IMPLEMENTATION DETAILS

In this section, we examine parallelization, metaparameters, and heuristics applied in our implementation, and touch upon the techniques that help to reduce overhead.

### 5.1 Parallelization

Performance improvements through parallelization are orthogonal to those of single-threaded, hazard-adaptive operators. However, parallelization incurs additional costs in two areas: sub-optimal load balancing and the merging of results produced by each thread.

Since the time taken to process a fixed number of tuples is variable, the input cannot simply be split equally by the number of worker threads as this could lead to poor load balancing, and consequently sub-optimal thread utilization. Therefore we implemented a task queuing system following the morsel-driven design [20].

Results from each worker thread must be merged and moved to their final output location. We use a hierarchical merging of results to maximize the parallelization of merge operations. Pairs of results are repeatedly merged in parallel until the final unified result is produced. We use a notification-based system to define the merge order. Worker threads notify the master thread when they have finished which generates a merge thread as soon as two sets of results are available, thus maximizing CPU utilization. We also apply operator-specific optimizations to the merge operation. For `SELECT`, worker threads construct results directly in the final output location, as each tuple's operation is independent. For `GROUP BY`, the merge operation is completed in a single pass by pre-sorting the results in the worker threads prior to merging.

### 5.2 Metaparameters and Trade-Offs

A key metric of adaptive algorithms is the maximum rate of change of the input data that the algorithm can adequately adapt to. This metaparameter, which we term the "adaptive period", is a necessary condition of adaptive algorithms and determines many implementation design choices. Naturally, the required adaptive period depends on the input data. Thus implementing an adaptive algorithm with a static adaptive period to work effectively across a range of datasets requires considering a number of associated trade-offs.

All of the switching mechanisms and tests discussed in Sections 3.4 and 3.6 incur an overhead. This overhead is amortized over the tuples processed between tests. Therefore the frequency of reading performance counter values, comparing high-level characteristics to tipping point thresholds, or performing probes of the hazard-affected algorithm is always a trade-off of adaptive responsiveness (smaller adaptive period) versus overhead. Consequently, the frequency of these tests should be set to achieve the desired adaptive period (responsiveness) whilst bearing in mind the additional overhead incurred. A similar trade-off also occurs in load balancing for multi-threaded implementations. Larger task sizes improve adaptivity effectiveness as the adaptive process is restarted less frequently. However, this comes at the cost of potentially worse load balancing.

[7]In homogeneous data the defining data characteristics remain consistent throughout the dataset, whereas for heterogeneous data they change throughout the dataset.

### 5.3 Heuristics

The performance of hazard-robust algorithms is not entirely constant, we found that this is primarily due to two factors. First, where the size of the output is variable it naturally affects performance. Second, multi-pass algorithms include step changes in performance depending on the number of passes required. We use heuristics to account for these factors by adjusting the required frequency of events necessary for the monitor to observe before it sends a signal to the dispatcher to increase hazard robustness. For example, adjusting it based on the selectivity/cardinality of the data, or by scaling it by the number of passes required for multi-pass algorithms (e.g., determined by the most significant bit for radix sort) where this differs from the value used to create the initial heuristic.

However, in our experiments, we found that a simple fixed value still closely followed the optimal static implementation even for real-world data. This simplification is feasible due to the consistency of the crossover point across various inputs and the only minor impact of different output sizes.

### 5.4 Engineering

Minimizing the overhead associated with a micro-adaptive operator requires careful engineering. Here we highlight key design choices.

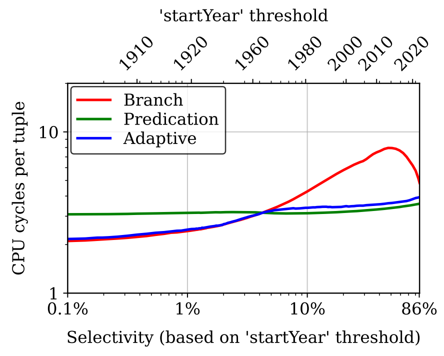
As highlighted in Section 3.5, hazard-affected algorithms should have a constant initialization cost. This requires lazily initialized data structures. In these cases, memory can be allocated using `calloc` and elements lazily initialized using the C++ "placement `new`" operator which separates initialization from allocation.

Merging results is a significant factor in the overall overhead of hazard-adaptive operators. Merge operations are required at two points: to merge the results of foundational algorithms when switching to a different one, and to merge results from worker threads in a multi-threaded implementation. In both cases, the merge operation is operator-specific. For example, the hazard-robust algorithm processes unprocessed micro-batches and combines these with the results from the hazard-affected algorithms. Depending on the operator this may be optimally achieved by adding the results to the unprocessed micro-batches or by using a separate merge process.

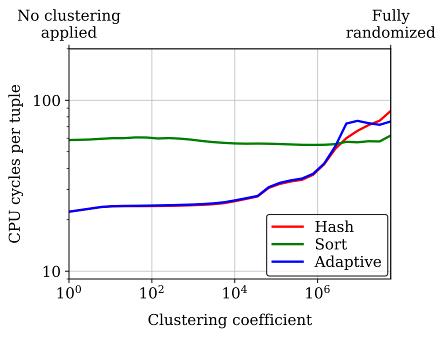
## 6 EVALUATION

### 6.1 Setup

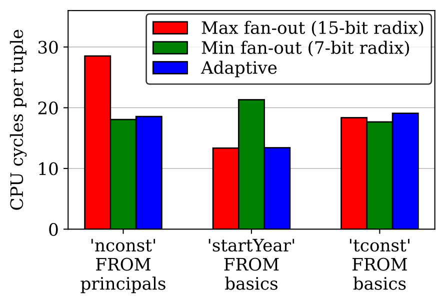
We evaluated our hazard-adaptive operators on a conventional server with a 4-core (physical and logical) Intel Xeon E3-1240 V2 running at 3.4 GHz. This processor has a 32 KB L1I and L1D cache, a 256 KB L2 cache, and an 8 MB shared L3 cache. We limited the machine to 4 KB pages for which the sTLB (L2 TLB) has 512 entries.



**(a) `SELECT startYear FROM basics WHERE startYear` ≤ `threshold`**



**(b) `SELECT titleID, COUNT(personID) FROM principles GROUP BY titleID`**



**(c) `PARTITION` performance on various columns from IMDB dataset**

**Figure 3: Performance benchmarks on real-world IMDB data**

The machine ran Ubuntu 18.04.6 with kernel version 4.15.0-96-generic and all benchmarks were compiled on g++ version 7.5.0 with the flags '`-O3 -march=native -std=c++17`'.

Single-threaded benchmarks are evaluated with the metric 'CPU cycles per tuple' as an indicative measure of throughput since it mitigates any potential impacts due to CPU frequency scaling and assists in fine-grained performance analysis at the instruction level. Normalization by the total number of tuples provides a performance metric that is meaningful across different-sized input datasets. Multi-threaded benchmarks use 'wall clock time' as there is not a single thread of execution on which to measure CPU cycles. Both metrics are measured via the PAPI API [27].

We evaluate performance on real-world data using the IMDB dataset [17]. Since we optimize operators in isolation we do not require a workload and instead benchmark operators individually.

All synthetic data benchmarks are performed using 200M 32-bit integers for both keys and payloads. All benchmarks were also repeated with 64-bit integers with no notable differences.

## 6.2 Real-World Data

Figure 3 benchmarks the three operators using the IMDB dataset.

Figure 3a shows the performance of the static and adaptive `SELECT` operators. Our hazard-adaptive approach matches the optimal static implementation with a small overhead due to the periodic checking of performance counters and, when running the hazard-robust algorithm, probes of the hazard-affected algorithm.

Figure 3b illustrates the performance of our adaptive `GROUP BY` aggregate operator. To assess robustness to auto-correlation/clusteredness we apply varying degrees of clusteredness. Such clusteredness would, for example, stem from data ingestion in regular intervals (longer intervals would lead to less clustering).

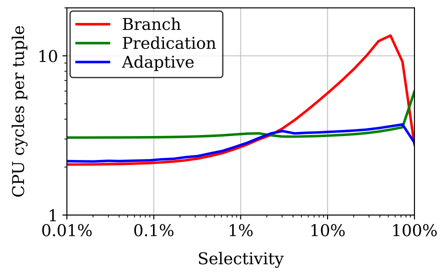
The clustering order is defined by specifying the number of tuples across which each value can be shuffled (the clustering coefficient in Figure 3b). A bucket is then formed for each window which covers this number of tuples but also stays within the bounds of the array. Each tuple is then randomly assigned one of the buckets covering that tuple. For example, for a clustering of three, the third tuple is randomly assigned to either the first, second, or third bucket. Following this, the tuples in each bucket are randomly shuffled. Finally, all the buckets are re-inserted into the original array to form the clustered version of the original data.

We observe a higher overhead when switching to the (sort-based) hazard-robust algorithm, observed as a step change in performance at the tipping point. This overhead is due to the high cardinality of the data relative to the total number of tuples. The high cardinality requires the initialization of a large hash table by the hazard-affected algorithm. However, this hash table is not used by the hazard-robust algorithm. This overhead could be reduced through the use of a lazily initialized hash map with $O(1)$ initialization costs. However, we leave the implementation of this for future work.

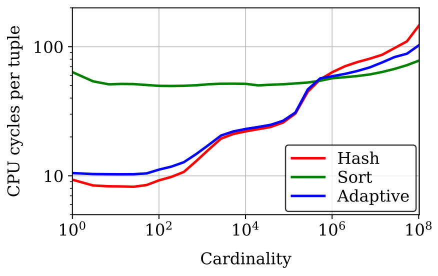
Figure 3c illustrates the performance of the hazard-adaptive `PARTITION` operation compared to the static implementations (maximum and minimum fan-outs) for three columns from the IMDB dataset. The first column ('`nconst`') is almost entirely random, and therefore the minimum fan-out algorithm provides the optimal performance. The second column ('`startYear`') has a sufficient degree of sortedness to prevent TLB thrashing even at the maximum fan-out. Consequently, the higher fan-out provides superior performance since it requires fewer recursive partitioning calls. The third and final column ('`tconst`') is both unique and fully sorted. However, only a single pass for all radix bit values is required to partition the data. Therefore, given that the input is sorted, all three algorithms complete a single pass without TLB thrashing. This means that the performance is similar across all implementations.

## 6.3 Synthetic Data

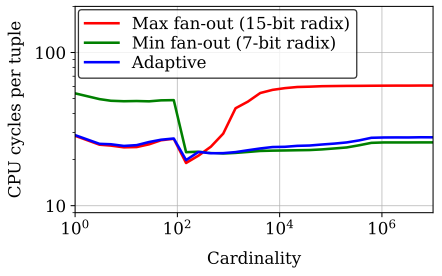
*Homogeneous Data.* Figure 4 shows operator performance on homogeneous data i.e., input datasets where the defining data characteristics remain consistent throughout the dataset. Figures 4a, 4b, and 4c illustrate the performance when varying the selectivity for `SELECT`, and cardinality for `GROUP BY` as well as `PARTITION` (higher-level order-invariant data characteristics). Figure 4d, 4e, and 4f show operator performance when varying the degree of randomness (i.e., % of randomly shuffled data in a sorted input) for `SELECT`, and clustering coefficient to represent variability for `GROUP BY` as well as `PARTITION` (order-dependent key data characteristics).



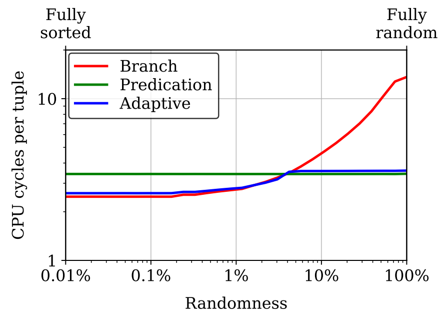
**(a) SELECT_Indexes**



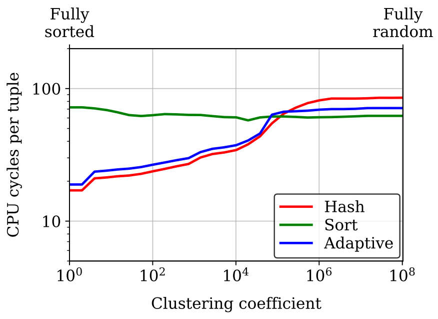
**(b) GROUP BY**



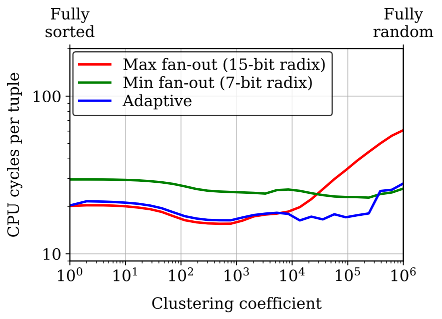
**(c) PARTITION**



**(d) SELECT_Indexes (50% selectivity)**



**(e) GROUP BY (10M cardinality)**



**(f) PARTITION (1M cardinality)**

**Figure 4: Operator performance on homogeneous data**

The performance of the hazard-robust algorithm in green is generally constant. The exception to this is the minimum fan-out algorithm in Figure 4c. In this case, we use a fixed maximum value of 20M and spread values between 1 and the maximum value. At low cardinality inputs, due to the large number of repeated small values, an additional pass is required by the minimum fan-out algorithm to partition the data (data is partitioned starting from the most significant bits). This results in a "step-shaped" improvement in performance at a cardinality of $\sim 10^2$ when a single partitioning pass becomes sufficient. We also observed this behavior in real-world data e.g., when partitioning the `startYear` column in Figure 3c.

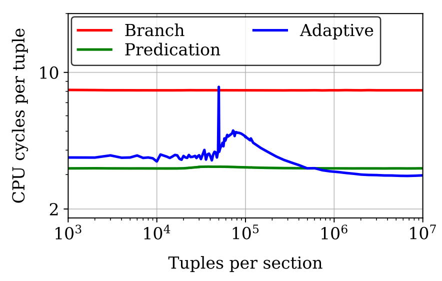
Furthermore, we observe that the hazard-robust algorithm performance in Figure 4f generally improves with increased clustering (more randomness). This also holds for the maximum fan-out implementation, where the performance improves with increased randomness until TLB thrashing begins to occur. We suspect that this is due to improved utilization of write-combine buffers [16, 34].

In all cases, the hazard-adaptive operator approach matches the optimal static implementation. However, the overhead associated with the GROUP BY adaptive operator is larger compared to SELECT or PARTITION. This is for two reasons: Firstly, the overhead at higher cardinality values or when running the hazard-robust algorithm is due to the high initialization costs of the hash map as described in Section 6.2. Secondly, we found that inlining compiler optimizations were inconsistently applied to the algorithms we tested. Sometimes applying to both the hash-based algorithm and hazard-adaptive algorithm (as is the case in Figure 3b) and sometimes only applying to the hash-base algorithm (as is the case in Figures 4b and 4e). We leave further investigation of this phenomenon for future work.

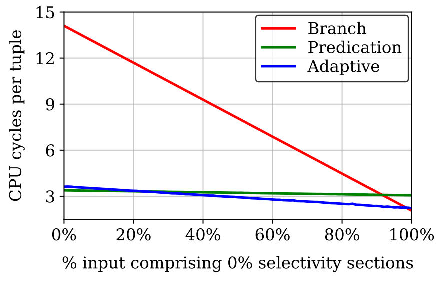
Across all but one of the experiments, the hazard-adaptive implementation follows the performance of the optimal foundational algorithm but does not exceed it. The exception is the PARTITION operation in Figure 4f. This is due to the hazard-adaptive PARTITION operation being comprised of a spectrum of hazard-affected algorithms (one for each radix bits value between the bounding values). Therefore, for some clustering values, the input is optimally processed using one of these intermediate radix bit values. When comparing the performance of the hazard-adaptive operator to the optimal radix bit value (the optimal static implementation with radix bit values between 7-15), we observe that it closely follows the optimal implementation. This demonstrates the benefit of using multiple hazard-affected foundational algorithms.

*Heterogeneous Data.* In Figure 5, we analyze hazard-adaptive operator performance when data characteristics vary within the input.

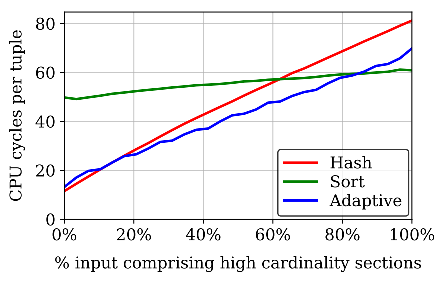
In Figure 5a the input data alternates between equal-length sections of high selectivity (50%) and low selectivity (0%). In this figure, we vary the number of tuples in each section and consequently, vary the number of sections in the input data since the total number of tuples is fixed at 200M. The performance of the foundational algorithms is independent of the number of tuples in each section because the overall average selectivity and predictability within the input remain constant i.e., only the frequency of switching selectivity is varied. We observe that when sections contain a large number of tuples the performance of the hazard-adaptive operator is superior to both of the foundational algorithms. This is because the hazard-adaptive operator can select the optimal foundational algorithm to process each individual section thereby achieving better performance than any static implementation.



**(a) `SELECT_Indexes`: Input alternating between equal-length sections of 50% and 0% selectivity**



**(b) `SELECT_Indexes`: Input alternating between 50% and 0% selectivity sections (varying relative number of sections)**



**(c) `GROUP BY`: Input alternating between high (10M) and low (100) cardinality sections (varying relative number of sections)**

**Figure 5: Operator performance on heterogeneous data**

Conversely, when there are a small number of tuples in each section the adaptive algorithm cannot work effectively as the input is changing more frequently than the adaptive period of the algorithm (50,000 tuples for our implementation of `SELECT`). For these inputs, the adaptive algorithm cannot adapt fast enough and will predominantly run the hazard-robust algorithm.

Between these two extremes, there is an increase in the CPU cycles per tuple. This rise occurs when the length of sections is on the order of the adaptive period. This causes the algorithm to frequently select the incorrect implementation to use for the next micro-batch. This, again, occurs because the input is varying too fast for the adaptive algorithm to respond to, leading to many tuples being processed using a sub-optimal implementation. The worst case occurs when the length of each section exactly equals the adaptive period. In this situation the adaptive algorithm is in anti-phase with the input, repeatedly executing the suboptimal algorithm. This results in worse performance than both foundational algorithms. However, the length of input sections needs to be almost identical to the adaptive period for this phenomenon to occur and is therefore mostly academic. Additionally, this phenomenon could be avoided entirely by randomly varying the monitoring interval.

In Figure 5b we fix the length of input sections (at a length that the hazard-adaptive operator is effective) and instead vary the number of 0% selectivity sections relative to the number of 50% selectivity sections. Therefore, when 0% of the input comprises 0% selectivity sections the input corresponds to a uniformly distributed input with 50% selectivity. At the other extreme, the input corresponds to a 0% selectivity input. Between these two extremes, the performance of all algorithms varies linearly as the relative number of sections is altered. The adaptive algorithm benefits from being able to optimally process the individual sections of the input and provides superior performance to both foundational algorithms when the benefit of adaptability outweighs the overhead.

Figure 5c evaluates `GROUP BY`, with sections varying between high and low cardinality. We make the same observations as for `SELECT`. However, unlike `SELECT`, the adaptive algorithm for `GROUP BY` uses a different number of tuples between checking the performance counters when executing the hazard-affected algorithm and between running probes when executing the hazard-robust algorithm. This is due to `GROUP BY` requiring a larger number of tuples in a probe to achieve an accurate last-level cache miss rate, and cannot switch algorithms based on the order-invariant metric of cardinality. This asymmetry in the length of the adaptive periods for the two underlying algorithms means that the true worst-case will occur when the length of each input section exactly matches the length of the adaptivity interval. Therefore, we omit this chart and consider a randomly ordered input of unique values as the realistic worst-case input (the right-hand side of Figure 4b).

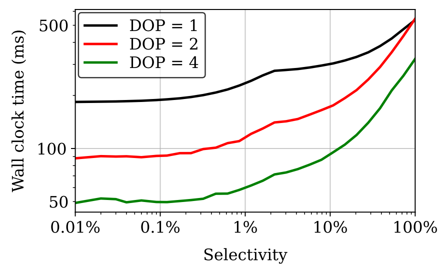
*Parallelism.* The benefits of hazard-adaptive operators are orthogonal to those from parallelism, however, it is important to evaluate the performance gain from parallelism given that real-world database processing is inherently multi-threaded. Figure 6 illustrates the multi-threaded performance of the `SELECT` and `GROUP BY` operators.

For `SELECT` in Figures 6a and 6b, the overhead of merging the results from individual threads is directly proportional to the selectivity which determines the number of elements to be moved to the output array. As a result, at low selectivity, the full benefit of parallelization is realized which reduces with increased selectivity.

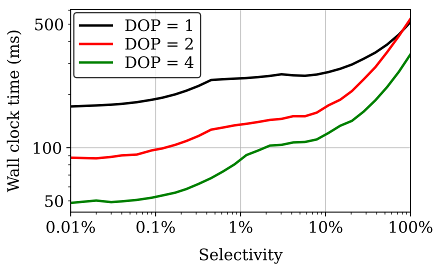
For the `SELECT_Values` variant of `SELECT` in Figure 6b the hazard-robust algorithm (utilizing vectorization and predication) becomes memory-bandwidth bound at $DOP = 4$. This has the effect of moving the tipping point to a higher selectivity value. We observe that this is captured in the heuristic (the rules within the monitor that determine when it is preferable to increase hazard-robustness) with the adaptive algorithm still correctly identifying the tipping point and switching implementation.

For `GROUP BY` in Figure 6c, the full benefit of parallelization is realized at low cardinality values (determines output size as well as initialization and merging overhead) and reduces as the cardinality increases. However, there is a discontinuity in the performance gain (performance improvement factor) between cardinalities of $\sim 10^4 - 10^6$, where increasing the DOP results in a smaller performance gain compared to the neighboring cardinality values.

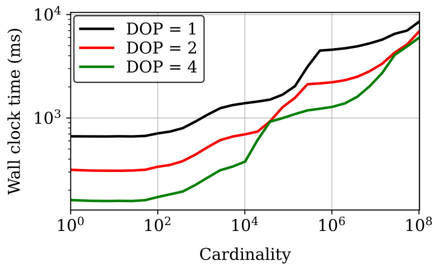
This drop in the performance gain is due to the sharing of finite resources, in this case, the L3 cache. Since the L3 cache is shared across cores, the effective cache size per thread decreases with increased DOP. This leads to the tipping point shifting to a lower cardinality value as the cache becomes full at a lower cardinality value. This shift in tipping point can be observed in Figure 6c and is what drives the reduced performance gain.



(a) `SELECT_Indexes`



(b) `SELECT_Values`



(c) `GROUP BY`

**Figure 6: Multi-threaded performance on homogeneous data varying higher-level order-invariant metric**

The reason for this shift in tipping points to lower cardinality values is subtle: Consider a single-threaded implementation of the hash-based algorithm that is making use of all the available cache, including the shared L3 cache. Increasing the DOP only increases the available L1 and L2 cache size, and therefore the performance increase cannot possibly be equal to the DOP increase. However, the sort-based algorithm has a much smaller cache footprint and does not require the use of the shared cache. Consequently, when running the hazard-robust algorithm the performance gain returns to the expected value. We can therefore conclude that the region between cardinalities of $\sim 10^4 - 10^6$ highlights the inputs at which the L3 cache is being heavily utilized when $DOP = 1$.

## 7 RELATED WORK

The classical approach to determining data characteristics, especially in the context of query optimization, involves the use of simple histograms usually built by sampling the input data [9]. However, this approach is not able to capture auto-correlation (patterns in the data). Approaches to determine order-dependent metrics prior to execution are complicated and brittle [1, 5, 14]. Our approach removes the need for analysis prior to execution, instead using the data itself to select the optimal operator implementation.

This approach of micro-adaptivity was introduced by Răducanu et al. [30] for the Vectorwise System (now Actian Vector). Răducanu et al. create a number of different operator implementations by applying a range of techniques (e.g., loop fission, loop unrolling, predication, vectorization) and using different compilers. At runtime, they employ a version of the multi-armed bandit (MAB) reinforcement learning algorithm [2, 7] to test each operator implementation during an "exploration" phase and select the optimal implementation during an "exploitation" phase. The MAB algorithm balances how resources are split between these two phases. Kaftan et al. expand upon this approach in Cuttlefish [19] and highlight the challenge of quickly and accurately selecting the optimal operator implementation with minimal overhead (~15% in the best-case scenario [30] and scales with the number of implementations [19]). We address these challenges through the use of hardware-assisted profiling with performance counters and simple heuristics.

Grizzly [15], a stream processing engine, employs performance counters for "adaptive optimizations". Fine-grained code instrumentation is used to identify optimizations and performance counters to trigger re-optimizations. Specifically, performance counters are used in two areas. One is to identify changes in predicate selectivity via branch mispredictions which triggers a profiling phase that incurs a ~50% drop in throughput. The other is to identify the use of a shared versus independent hash map in a multi-threaded implementation using exclusive accesses to a cache line that another thread also accesses exclusively to detect data skew. This includes using a histogram to assess the potential benefit of a shared hash map. In our approach we remove profiling phases entirely, making optimization decisions purely from performance counter values.

In contrast, Zeuch et al. [36] use performance counters without any code instrumentation to drive progressive optimization [23] by triggering a re-optimization process based on runtime comparisons of performance counter readings against cost models [36]. Their use of performance counters incurs a negligible overhead, unlike a comparable instrumentation approach (~100% overhead). Unlike in our work, Zeuch et al. focus on the overall query level thus requiring re-optimization events. Additionally, the requirement for accurate cost models limits the extensibility. By limiting our hazard-adaptive approach to individual operators we can provide a generalized framework and track the optimal static implementation with only simple heuristics and switching logic.

## 8 CONCLUSION

Micro-architectural hazards are one of the key challenges for in-memory data processing systems. While hazard-robust implementations exist for many operators, their performance is worse in the general case. As the performance impact depends on auto-correlation in the data and varies within a table, static query optimization is ill-suited to solve this problem.

To address this challenge, we proposed to select the hazard-optimal operator at runtime. We defined a conceptual framework for hazard-adaptive operators. We applied this framework to implement a number of hazard-adaptive operators in a prototypical relational data processor, demonstrating that this hazard-adaptive approach provides robustness to a range of data characteristics and even adapts to changing patterns within the data.

In the future, we will build on the framework to implement and compare other micro-adaptivity approaches. We also plan to extend our approach to include optimizations for factors beyond micro-architectural hazards. The long-term objective of this line of research would be the fully-dynamic optimization of operators and even full query plans.

## REFERENCES

[1] Miklós Ajtai, T. S. Jayram, Ravi Kumar, and D. Sivakumar. 2002. Approximate Counting of Inversions in a Data Stream. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '02)*. Association for Computing Machinery, New York, NY, USA, 370–379. https://doi.org/10.1145/509907.509964

[2] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47 (2002), 235–256.

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[4] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. *Proceedings of the 2021 International Conference on Management of Data* (2021). https://api.semanticscholar.org/CorpusID:231978821

[5] Sagi Ben-Moshe, Yaron Kanza, Eldar Fischer, Arie Matsliah, Mani Fischer, and Carl Staelin. 2011. Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation. In *Proceedings of the 14th International Conference on Database Theory* (Uppsala, Sweden) *(ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/1938551.1938584

[6] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal* 27 (2018), 797–822.

[7] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. 2012. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning* 5, 1 (2012), 1–122.

[8] Pedro Celis, Per-Ake Larson, and J. Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 281–288. https://doi.org/10.1109/SFCS.1985.48

[9] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1998. Random Sampling for Histogram Construction: How Much is Enough?. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 436–447. https://doi.org/10.1145/276304.276343

[10] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

[11] Andrew Crotty, Alex Galakatos, and Tim Kraska. 2020. Getting Swole: Generating Access-Aware Code with Predicate Pullups. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1273–1284. https://doi.org/10.1109/ICDE48307.2020.00114

[12] Ulrich Drepper. 2007. What every programmer should know about memory. *Red Hat, Inc* 11, 2007 (2007), 2007.

[13] Stéphane Eranian. 2008. What Can Performance Counters Do for Memory Subsystem Analysis?. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: Held in Conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)* (Seattle, Washington) *(MSPC '08)*. Association for Computing Machinery, New York, NY, USA, 26–30. https://doi.org/10.1145/1353522.1353531

[14] Parikshit Gopalan, T.s Jayram, Robert Krauthgamer, and Ravi Kumar. 2007. Estimating the sortedness of a data stream. 318–327. https://doi.org/10.1145/1283383.1283417

[15] Philipp M. Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2487–2503. https://doi.org/10.1145/3318464.3389739

[16] Seokbin Hong, Won-Ok Kwon, and Myeong-Hoon Oh. 2020. Hardware Implementation and Analysis of Gen-Z Protocol for Memory-Centric Architecture. *IEEE Access* 8 (2020), 127244–127253. https://doi.org/10.1109/ACCESS.2020.3008227

[17] IMDb. 2023. IMDb Non-Commercial Datasets. Available from: https://developer.imdb.com/non-commercial-datasets/. Accessed: 22/08/2023.

[18] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. 1997. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 618–625.

[19] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. arXiv:1802.09180 [cs.DB]

[20] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[21] Corey Malone, Mohamed Zahran, and Ramesh Karri. 2011. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. In *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing* (Chicago, Illinois, USA) *(STC '11)*. Association for Computing Machinery, New York, NY, USA, 71–76. https://doi.org/10.1145/2046582.2046596

[22] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. on Knowl. and Data Eng.* 14, 4 (jul 2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210

[23] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust Query Processing through Progressive Optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 659–670. https://doi.org/10.1145/1007568.1007642

[24] S. McFarling and J. Hennesey. 1986. Reducing the Cost of Branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (Tokyo, Japan) *(ISCA '86)*. IEEE Computer Society Press, Washington, DC, USA, 396–403.

[25] Subrata Naskar and J. P. Naveen. 2020. *Histograms based on varying data distribution*. Hewlett Packard Enterprise Development LP. https://patents.google.com/patent/US10628442B1/en

[26] Stratos Idreos Fabian Groffen Niels Nes and Stefan Manegold Sjoerd Mullender Martin Kersten. 2012. MonetDB: Two decades of research in column-oriented database architectures. *Data Engineering* 40 (2012).

[27] The University of Tennessee. 2023. *PAPI: Performance Application Programming Interface*. Innovative Computing Laboratory. http://icl.utk.edu/papi/

[28] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

[29] Kenneth A. Ross. 2004. Selection Conditions in Main Memory. *ACM Trans. Database Syst.* 29, 1 (mar 2004), 132–161. https://doi.org/10.1145/974750.974755

[30] 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)*. Association for Computing Machinery, New York, NY, USA, 1231–1242. https://doi.org/10.1145/2463676.2465292

[31] Tessil. 2023. *Tessil/robin-map*. https://github.com/Tessil/robin-map MIT License.

[32] Gary Scott Tyson. 1994. The Effects of Predicated Execution on Branch Prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (San Jose, California, USA) *(MICRO 27)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/192724.192753

[33] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. 2008. Exploiting Hardware Performance Counters. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. 59–67. https://doi.org/10.1109/FDTC.2008.19

[34] Jan Wassenberg and Peter Sanders. 2010. Faster Radix Sort via Virtual Memory and Write-Combining. arXiv:1008.2849 [cs.DS]

[35] Vincent M. Weaver and Sally A. McKee. 2008. Can hardware performance counters be trusted?. In *2008 IEEE International Symposium on Workload Characterization*. 141–150. https://doi.org/10.1109/IISWC.2008.4636099

[36] 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

[37] Zuyu Zhang, Harshad Deshmukh, and Jignesh M Patel. 2019. Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities.. In *CIDR*.

[38] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/564691.564709