White-Box Micro-Adaptive Query Processing

Jack Pearce
Imperial College London
jack.pearce22@imperial.ac.uk

Hubert Mohr-Daurat
Imperial College London
h.mohr-daurat19@imperial.ac.uk

Holger Pirk
Imperial College London
hlgr@ic.ac.uk

***Abstract*—Operator performance in in-memory data management systems (DMS) 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 order-dependent data characteristics that query optimizers struggle to capture (e.g., sortedness, clusteredness) making a priori hazard-conscious optimization difficult. Additionally, statically optimized plans fail to adapt when data characteristics vary within a table. To address these problems, we propose a hazard-adaptive approach to query execution. Through hardware-assisted runtime profiling of low-level metrics, operators dynamically adapt to "hazardous" data. We propose an architecture for hazard-adaptive operators and integrate our approach into a DMS. We demonstrate that using hazard-adaptive operators provides a ∼2-20× speedup across several TPC-H queries.**

I. Introduction

Analytical query performance depends on queries and data characteristics. Queries define the operations and the optimizations that can be applied. Data characteristics determine the performance of these operations. As the number of operators is usually moderate, capturing query characteristics (e.g., selection/join predicates or grouping keys) of a dataflow execution engine is comparatively easy. Accurately formalizing data characteristics, however, is significantly harder.

Approaches for capturing data characteristics are typically tailored to logical query optimization, i.e., the order of relational operators within a query. The data characteristics for this do not capture the ordering of the data[1] making them order-invariant. For example, simple histograms (usually built on sampled values) suffice to estimate single-attribute cardinalities for join or selection-order optimization. To capture complications like correlation, histograms are multidimensional and samples larger, though the principle remains the same.

However, query processing performance is also highly dependent on order-dependent metrics i.e., patterns in the data (e.g., sortedness, clustering, auto-correlation). This includes the selection of the optimal (physical) algorithm to implement a logical operator. For example, Figure 1 shows the effect of sortedness on the performance of two implementations of the selection operator: one branching and one branch-free [1]. The data is at a fixed selectivity with only the order varied. Despite the branch-free implementation leveraging SIMD the branching version outperforms it at low randomness (increasing with parallelization as memory bandwidth becomes a constraint).

[1]The notable exception to this is the maintenance of "interesting orders" for opportunistic use of a merge-join or merge-aggregation optimization.

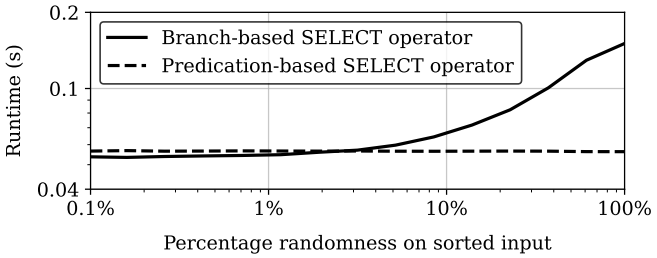


Fig. 1. Runtime of `SELECT` index generation at a fixed 50% selectivity for 25M 64-bit integers on a conventional server

Overall, data sortedness can have a 3× performance impact on the selection operator (we found up to 10× on other machines). We found other operators (e.g., group, join, intersect) suffer similar problems. These data patterns or localities are prevalent in real-world data. For example, the 'basics' IMDB table [2] is highly clustered (but not sorted) by 'startYear'. Climate and sensor data [3] exhibit similar locality.

Capturing these patterns in the data is challenging. As histograms are order-invariant, they do not capture auto-correlation, and extending them to do so is far from trivial [4]. Other approaches to measuring the sortedness of data tend to be complicated and brittle [5]–[7]. Any approach based on these techniques would inherit these traits and thus optimizers relying on this information risk making wrong decisions.

An alternative approach would be to not quantify auto-correlation a priori, but implement operators that adapt their behavior at runtime on a batch-by-batch basis. This is conceptually related to the paradigm of "micro-adaptivity" which Răducanu et al. [8] introduce as "a framework that keeps many alternative function implementations in a system". This addresses two problems: first, it makes the operator robust against sub-optimal optimizer decisions. Second, micro-adaptive operators change their implementation to make the best use of regions of locality within a single dataset which addresses a known shortcoming of classic optimizers.

Răducanu et al. propose a black-box micro-adaptive approach to select the optimal assembly code implementation at runtime: a test and learn framework using execution time to capture utility/reward [8]. This approach relies on a common operator interface, abstracting implementation details. However, this requires all implementations to be continuously tested to select the best; limiting this approach to simple cases (e.g., using different compilers). Furthermore, if data locality varies within a table, this must be identified and the adaptivity-algorithm restarted, resulting in high overhead [8], [9].

To address these shortcomings, we propose micro-adaptive operators which are "white-box" in a system that continuously monitors the associated micro-architectural hazards and "indicates" when a switch to an alternative implementation is preferable. We make the following technical contributions:

- We present a hazard-adaptive data management system architecture, including hazard-adaptive white-box operators that maintain state across operator implementations.
- We develop a class of algorithms that are hazard-adaptive versions of classic data-processing algorithms. We exemplarily apply this to three operators: `SELECT`, `GROUP`, and `JOIN`. This includes a simple, elegant, effective extension of radix partitioning that utilizes sideways information passing between the input tables to achieve adaptivity based on the locality in both tables.
- To demonstrate practicality we integrate the hazard-adaptive operators into a database management system to act as a virtually overhead-free kernel accelerator. We evaluate the performance across a range of TPC-H queries and observe $\sim$2-20$\times$ performance improvements.

We cover background knowledge in Section II, present the hazard-adaptive architecture in Section III, and develop concrete operators in Section IV. In Section V we extend the architecture to co-adaptive operators, and Section VI presents the system architecture. We evaluate performance, discuss related work, and conclude in Sections VII, VIII and IX.

II. Background

This section provides the background for hazard-adaptivity.

A. Micro-Architectural Hazards

Micro-architectural hazards are defined as "situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle" [10]. To develop high-performing, micro-architecturally efficient algorithms, it is crucial to minimize hazards. This minimizes stall cycles and maximizes pipeline utilization. The classes of hazards are:

(1) Control hazards are caused by control-flow changes (branches or jumps) leading to stall cycles and pipeline flushes due to bad speculation. Stall cycles can also be caused by a lack of eligible instructions due to control-flow dependencies.

(2) Data hazards occur when the instruction pipeline stalls due to operands not being ready at execution. Such stalls result from cache or TLB misses when data retrieval from main memory is slower than the CPU's processing speed.

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

B. Micro-Architecturally Efficient Algorithms

Micro-architecturally efficient algorithms maximize CPU efficiency by minimizing hazards. The principles underpinning these algorithms fall into four categories [10]:

(1) Data Locality and Cache Awareness: Maximizing cache utilization minimizes data hazards and prevents stalls. This is achieved by increasing data locality to prevent unnecessary cache misses, increasing cache line utilization, reducing the program cache footprint, and prefetching 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 caused by a lack of eligible instructions.

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

(4) Optimizing Memory Access Patterns: A high cache miss rate is typically due to a random memory access pattern and can be reduced by switching to a sequential access pattern.

Performance engineering techniques apply these principles to increase an algorithm's robustness to hazards and thereby develop micro-architecturally efficient algorithms.

C. Hardware Performance Counters

Performance counters are a set of registers within the CPU performance monitoring unit (PMU). These counters can be configured to count micro-architectural events during program execution, enabling the profiling of software [11]–[13]. This makes them a crucial resource for identifying and solving performance issues [14]. The key advantage of performance counters lies in their near zero execution cost [12], [15], [16].

III. Hazard-Adaptive Architecture

As shown in Section 1, the optimal operator implementation depends on input data characteristics due to hazards. A micro-adaptive operator should use the frequency of these hazards to select the optimal implementation. Thus, the objective is to design *hazard-adaptive* operators that switch between implementations at runtime based on changing hazard rates.

Figure 2 depicts the architecture we used to achieve this hazard-adaptive behavior. The hazard-adaptive operator interface matches a non-adaptive operator but also accesses performance counters to inform hazard-adaptivity. The foundation of the hazard-adaptive operator is a number of alternative operator implementations: the "foundational algorithms".

The foundational algorithms are coordinated and orchestrated by a "dispatcher" which selects the optimal algorithm to use, i.e., the active operator." It dispatches micro-batches of the input vector to the active operator, combining the foundational algorithms into a single hazard-adaptive operator.

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. In the subsequent sections, we will discuss each component in Figure 2 in detail.

A. Hazard-Affected (HA) Algorithms

Foundational algorithms form the options of alternative implementations that the dispatcher can select. To create these

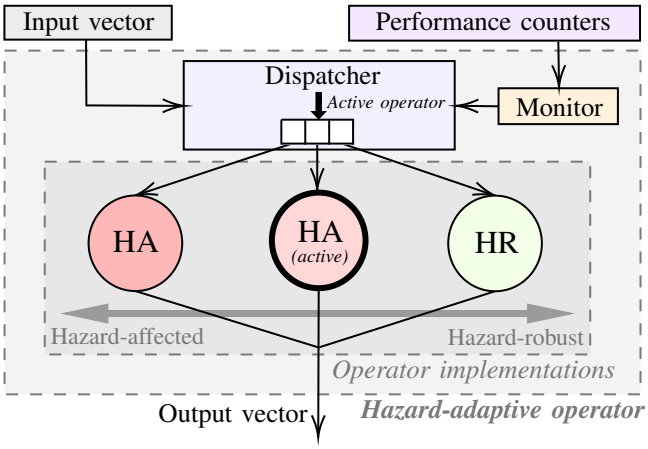


Fig. 2. Hazard-adaptive operator architecture schematic

alternative implementations we distinguish two categories of algorithms, "*hazard-affected*" and "*hazard-robust*" algorithms. Hazard-adaptive operators require a single hazard-robust algorithm and at least one hazard-affected algorithm.

The defining objective of hazard-affected algorithms is the minimization of work (typically 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 requires it, making the number of instructions executed dependent on the data. Secondly, this frequently introduces control-flow dependencies (to determine whether to execute work) making the algorithm susceptible to control hazards. Moreover, algorithms within this category almost universally do not focus on micro-architectural efficiency, instead focusing on work efficiency. Consequently, they are susceptible to the micro-architectural hazards described in Section II-A, the generation of which is also determined by the data characteristics. This dependency of performance on the data characteristics makes the performance of hazard-affected algorithms hard to predict.

To illustrate this, consider a branching implementation of the `SELECT` operator: `if(/*condition*/)output[j++] = input[i++];`. It has highly variable performance for both of these reasons. The output is only generated if the selection predicate is satisfied, and the performance at a given selectivity is determined by the sortedness of the input due to control-flow dependencies and the resulting control hazards. The variable performance is illustrated in Figure 1 and means that a hazard-robust algorithm is superior under certain data characteristics

B. Hazard-Robust (HR) Algorithms

The alternative category of algorithms are hazard-robust. Unlike hazard-affected algorithms, the key principle of hazard-robust algorithms is micro-architectural robustness, i.e., limited exposure to control, data, and structural hazards.

The performance of hazard-robust algorithms is far less dependent on the input data versus hazard-affected algorithms. Their robustness to hazards ensures consistently high CPU instruction pipeline throughput, i.e., largely unaffected by data characteristics. Additionally, as we will illustrate in Section VII, hazard-robust algorithms are typically designed such that the quantity of work they perform is largely independent of data characteristics. This is illustrated by the predication-based implementation of the `SELECT` operator: `output[j] = input[i++]; j += (/*condition*/);`. Figure 1 shows that its performance is independent of the data sortedness.

There is substantial work on techniques to develop hazard-robust versions of hazard-affected algorithms [1], [17]–[22]. However, existing data management systems typically do not include hazard-robust implementations because they sacrifice best-case performance for worst-case performance. Hazard-adaptive operators provide an architecture to make these algorithms useful by executing them only when it is beneficial.

C. The Monitor

To maximize best-case performance the active operator starts as the most hazard-affected implementation.

Dominant Micro-Architectural Event: To identify when to switch to a more robust algorithm the performance of the hazard-affected algorithm must be quantified. This can be achieved using the hazard frequency, as hazards degrade performance. To quantify performance with a single hazard, we define the dominant micro-architectural event as the primary source of hazards degrading the algorithm's performance.

By periodically sampling the event counter and calculating the per-tuple frequency of the dominant event, the monitor assesses the current hazard-affected algorithm performance.

A key advantage of this performance monitoring over traditional code instrumentation is that it monitors the key data characteristic driving performance rather than a higher-level characteristic that provides only a partial view. For example, in Figure 1 randomness (more precisely, predictability) is the key data characteristic, not simply selectivity. Using performance counters directly captures these patterns in the data.

Heuristics: To signal the dispatcher when to switch algorithms, the monitor requires a heuristic to determine the threshold event frequency where increased robustness is beneficial. This threshold is influenced by hardware, element size, and the degree of parallelization (DOP). Thus, a unique heuristic is needed for each combination of these factors on each machine.

These heuristics are static and therefore can be considered as machine constants. This is because many micro-architectural events are per core (e.g., branch mispredictions, TLB misses), isolating them in concurrent scenarios. However, when relying on a shared resource, the algorithm's adaptivity must account for resource contention as this impacts performance. The shared nature of performance counters makes these heuristics inherently adaptive to this contention. For example, algorithms with last-level cache (LLC) misses as the dominant event (e.g., hash aggregation) will account for other processes using the LLC, and the monitor will increase robustness accordingly.

D. Dispatcher

Hazard-Adaptive Operator Interface: To ensure extensibility, we decouple the heuristics for identifying the threshold hazard frequency (the monitor) from the switching logic used to select the active implementation (the "dispatcher"). This

```
enum Event {BranchMispredict, CacheMisses, TlbMiss, ...};

template <class InputType, class OutputType>
class HazardAdaptiveOperator {
  HazardAdaptiveOperator(const vector<InputType> &input);
  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 inputSz, size_t maxOutSz);
  void updateState(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

allows the monitor to track and respond to changes in multiple micro-architectural events. This allows for extension to more complex operators requiring tracking of multiple hazards.

To communicate with the dispatcher the monitor uses the push-based hazard-adaptive operator interface in Listing 1, using `adjustRobustness()` to signal when increasing robustness is beneficial. Based on `adjustment` value, the dispatcher selects the active operator and continues to process micro-batches via the push-based interface in Listing 2.

Foundational Algorithm Interface: To avoid repeated initialization costs and ensure efficient use of intermediate data structures, foundational algorithms are encapsulated into an object that maintains these resources. For example, allocated memory (e.g., hash table) is reused for all micro-batches processed by that operator implementation. Listing 2 defines the object interface which the dispatcher interacts with.

This interface allows the dispatcher to retain previous processing across algorithm switches efficiently. This is a particularly challenging aspect of micro-adaptivity when applied to more complex operators such as pipeline-breakers which require information sharing across the entire input dataset. To retain progress across switches, results from different algorithms must be merged. Since the switch is made to the hazard-robust algorithm (prompting the merge), some data isn't suitable for a hazard-affected algorithm. Thus, the hazard-robust algorithm handles the merge. If results are merged immediately after a switch, some work is duplicated by merging the same results repeatedly. To avoid this, merges are deferred until the entire input is processed.

To efficiently merge results, each foundational object has an internal state that is passed (copy-free) from the hazard-affected implementations to the hazard-robust implementation during a merge. This internal state, collating results from previous `processMicroBatch()` calls, is extracted by the dispatcher using the `getState()` method and transferred to another implementation via the `updateState()` method.

At construction of the hazard-affected algorithm objects, the dispatcher does not know how many tuples will be processed with each object. Therefore, the initialization cost must be $\mathcal{O}(1)$, and any memory allocations must be lazily initialized to ensure that initialization costs are only incurred if the memory is used. However, 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 the hazard-robust algorithm will process. Therefore, the initialization cost can be $\mathcal{O}(inputTuples)$.

Decreasing Robustness: In addition to increasing robustness, a hazard-adaptive operator must also be able to decrease robustness to adapt to repeated changes in data characteristics.

When a hazard-affected algorithm is executed and generates few hazards, the monitor can signal the dispatcher to switch to a more hazard-affected algorithm by using a negative adjustment value in 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) For operators 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 in which it is preferable to decrease robustness (e.g., 0% selectivity implies full predictability).

(2) The dispatcher can periodically perform probes (tests) with a hazard-affected algorithm which will switch optimally. These probes should run the minimum tuples needed to obtain a statistically reliable dominant hazard frequency.

All hazard-adaptive operators use **(2)**, with **(1)** also used opportunistically when permitted by the operator's semantics.

IV. Hazard-Adaptive Operators

Creating hazard-adaptive operators is non-trivial, requiring selecting foundational algorithms, identifying dominant micro-architectural event(s), developing accurate heuristics for the monitor, and efficiently switching between implementations via the dispatcher. We develop three such operators and discuss their implementation in the proposed framework.

The problem space for hazard-adaptive operators is broad, encompassing various operators and implementations specific to each database management system. To validate the concept of hazard-adaptive operators, we examine `SELECT`, `JOIN`, and `GROUP`, which cover a substantial portion of this space.

A. Hazard-Adaptive `SELECT` Operator

Foundational Algorithms: A hazard-affected implementation of the `SELECT` operator is the branching algorithm: `if(/*condition*/)output[j++]=input[i++];`. Work is minimized by updating the output only if the conditional predicate is satisfied. However, conditionals can cause branch mispredictions (control hazards), leading to pipeline flushes [23].

To create an alternative branch-free implementation, predication can be applied by always copying to the output:

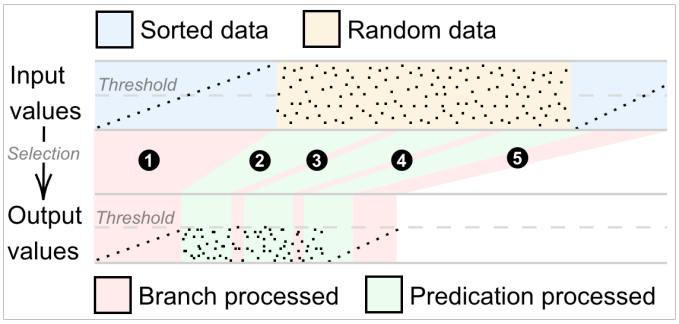


Fig. 3. Probing of hazard-affected SELECT algo. on varying locality data

`output[j]=input[i++]; j+=(/*condition*/);`. This will be auto-vectorized by a modern compiler. Predication makes the algorithm robust to control hazards but incurs extra work by updating the output before testing the predicate.

For the branch-free algorithm, the number of instructions per tuple is constant and the performance is 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 fails. These mispredictions dominate performance, as the resulting pipeline flushes create 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 metrics, as varying the predictability of data at fixed selectivity impacts the performance of the branching implementation due to mispredictions.

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

Dispatching Logic: Figure 3 shows how the SELECT operator processes an input composed of regions of high data locality (predictable) and low data locality (unpredictable). The dispatcher starts by processing tuples with the hazard-affected algorithm ❶ before switching to the hazard-robust branch-free algorithm when it receives a signal from the monitor ❷. 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 be superior (❸ to ❺).

In addition to running probes, the dispatcher contains logic for switching back to the branch-based algorithm based on the (order-invariant) metric of selectivity (e.g., at very low or high selectivity, the hazard-affected algorithm is always superior).

B. Hazard-Adaptive GROUP Operator

Foundational Algorithms: A hazard-affected implementation of the GROUP operator is a hash-based aggregation. Tuples are inserted into a hash table and work is minimized by aggregating a payload immediately with the corresponding existing aggregated payload in the hash table. The algorithm returns an unordered array of unique keys with their corresponding aggregated payload. This algorithm is susceptible to capacity cache misses (data hazards). Capacity cache misses occur due to a random memory access pattern on a data structure that exceeds the cache size. This leads to "cache thrashing" due to frequent data swapping into and out of the cache.

In the implementation, we used Tessil's `robin_map` [24] which utilizes "Robin Hood" hashing [25] and outperformed other hash tables[2]. We modify it to be lazily initialized ($\mathcal{O}(1)$ complexity) in line with the requirements in Section III-D.

An alternative hazard-robust algorithm must reduce exposure to data stalls, meaning a smaller cache footprint. One approach is to replace the random memory access pattern with a sequential one. This requires a multi-pass algorithm, allowing for data prefetching and improved cache line utilization. Radix sort is a multi-pass algorithm that can create a sort-based implementation. After sorting the input, a final pass aggregates adjacent tuples with matching keys. However, this incurs the cost of unnecessarily sorting the output.

The fan-out, defined as $2^{radixBits}$, which determines the number of partitions (and therefore typically pages) written to during the data shuffling phase of radix sort, must be below that of the sTLB[3] capacity to ensure that sTLB store misses (a data hazard) do not occur and the algorithm is hazard-robust.

Monitoring Heuristics: The dominant micro-architectural event for the hash-based algorithm is last-level cache misses, which are the cache misses that most affect performance.

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 is illustrated by inputs of equal cardinality and different clustering[4]. A clustered input means only a subset of the hash table will be repeatedly accessed at each point in the input. Therefore, clustering reduces the working set size (cache footprint), and a hash table larger than the cache can be used without cache thrashing.

Therefore the monitor heuristic is based on the frequency of last-level cache misses. However, a disadvantage is that a relatively large number of tuples is required to determine whether the working set size exceeds the last-level cache. Since the hash table is lazily initialized, with memory allocated by `calloc`[5], when pages are first updated they need to be allocated and initialized, increasing the cost of these initial writes. Therefore the cost is non-linear with a disproportionate cost for the initial tuples. This makes an efficient implementation of a hazard-adaptive GROUP operator challenging.

Identifying when to switch as early as possible is the only way to mitigate this cost. For the hash-based GROUP algorithm page faults can be used since the first time a page is touched it must be initialized (generating a page fault). The frequency of page faults gives a much quicker indication of the working set

[2]Google `dense_map`, Abseil `flat_map`, Facebook `F14_map`.

[3]Shared TLB, aka unified TLB (uTLB), or L2 TLB. Note this is per core.

[4]Defined as the similarity between adjacent values, e.g., clustering of 1,000 means adjacent values would be $\leq$ 1,000 indexes apart if the array were sorted.

[5]Allocated memory points to a single zeroed-out "copy-on-write" page.

size versus last-level cache misses when processing the first tuples. However, the page fault rate is transient. Therefore the heuristic must use a % of the input and be based on both the page fault frequency and the rate of decrease over a number of tuples. In other words, if the page fault rate is high and only decreases slowly, this means the working set size must be large and an increase in robustness is required.

Thus, the monitor not only requires a simple heuristic for last-level cache misses but also monitors the page fault rate to quickly identify when to switch in a subset of cases.

Dispatching Logic: For efficiency, the hazard-affected object reuses the hash table for all micro-batches processed. Additionally, the hazard-robust algorithm is deferred until after the hazard-affected algorithm has processed all the necessary micro-batches (of the three operators this out-of-order processing optimization is only required for `GROUP`). The sort-based algorithm processes the remaining micro-batches of the input in addition to the results collated in the hash table which forms the state of the hash-based implementation. This state is passed into the hazard-robust algorithm at construction using move semantics (Section III-D). At construction, the additional radix sort buffer can be eagerly initialized since the dispatcher knows the total number of tuples that will be processed.

We rely on an oracle for cardinality estimates and apply an industry-standard $2.5\times$ over-allocation factor. Creating a hash-based `GROUP` operator robust to cardinality estimates (analogous to `JOIN` [26]) is orthogonal to our work and can be incorporated by switching out the foundational algorithm.

As in Figure 3 the dispatcher runs probes of the hazard-affected algorithm. However, as the hazard-robust algorithm is deferred until after the hazard-affected algorithm the hazard-affected algorithm effectively jumps forward to run probes, leaving a block of input tuples for the hazard-robust algorithm.

V. Co-adaptivity

While effective for single-input operators, the hazard-adaptive architecture in Figure 2 is ill-suited for multiple-input table operators such as `JOIN`, `PARTITION` (on multiple tables), and `EXCEPT`. For example, consider a hazard-adaptive `INTERSECT` using a hash-based and a sort-merge approach. If the tables are processed sequentially starting with the hash-based approach, the first table may produce few cache misses, but the second table could generate many cache misses. This requires either switching to the sort-merge approach (discarding the hash table work) or suffering performance degradation from cache misses. This disjoint processing fails to capture the data characteristics of both tables simultaneously.

To address this we define a new class of micro-adaptive operators: *co-adaptive* operators. In these operators, tables are considered together and co-processed, adapting to all tables.

A. Co-Adaptive Architecture

To create an architecture for co-adaptive operators we extend the hazard-adaptive operator architecture in Figure 2, resulting in Figure 4. We introduce an "input interleaver" that marks input tuples and passes them to the hazard-adaptive

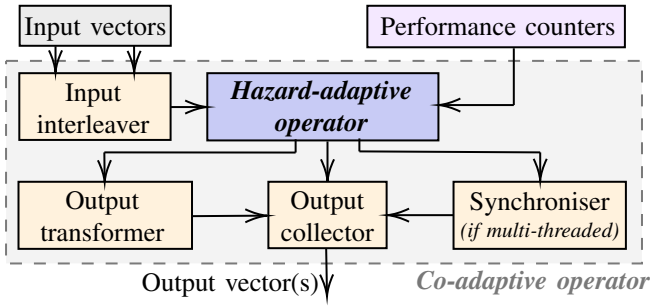


Fig. 4. Co-adaptive operator architecture schematic

operator as if they were a single table. The hazard-adaptive operator processes micro-batches from each table in turn, making the monitor's adaptivity decisions based equally on all tables. The information denoting the table that tuples belong to is propagated through the operator to the "output collector" which collates the results. This process is shown in Figure 5 for the `PARTITION` operator (discussed in Section V-B).

For co-adaptive operators where the foundational algorithms have different outputs (e.g., partitioning using different radix bits) there is the challenge of ensuring the results structure in the output collector is consistent with the active operator. To address this, an "output transformer" receives signals from the monitor when robustness changes and ensures consistency between the active operator and the output collector results.

When multi-threaded we include a "synchronizer" that synchronizes threads and ensures they are all at the same point in the adaptivity space. A global variable tracks the maximum robustness required by all threads. After every micro-batch, threads update this variable if necessary and then synchronize to the current value. The overhead is minimal as this process is lock-free since this variable is monotonic. However, periodic probes of the hazard-affected algorithm require a global synchronization event to reset the global robustness variable.

Returning to the `INTERSECT` operator example, using the co-adaptive operator architecture and hashing parts of each table in turn (outputting on matches) a switch to the sort-merge implementation can be made without losing progress.

B. Co-Adaptive `JOIN` (`PARTITION`) Operator

A hash-join suffers last-level cache misses when the build-side hash table (the smaller table) doesn't fit in the cache. The join can be made robust by partitioning the input tables, then joining partitions (which now fit in the cache) independently and unioning the results (known as a radix partitioned hash join [27]). Therefore, we focus on the `PARTITION` operation as this is the critical phase for making large table joins robust and has broad applications, forming the basis of many operators [28]. Limiting the `PARTITION` operator to `JOIN` makes it co-adaptive, as `JOIN` has two input tables. Thus, we use the co-adaptive architecture in Figure 4.

Foundational Algorithms: To minimize the work required for radix partitioning a large fan-out, defined as $2^{radixBits}$, can be used to partition data in a single 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.

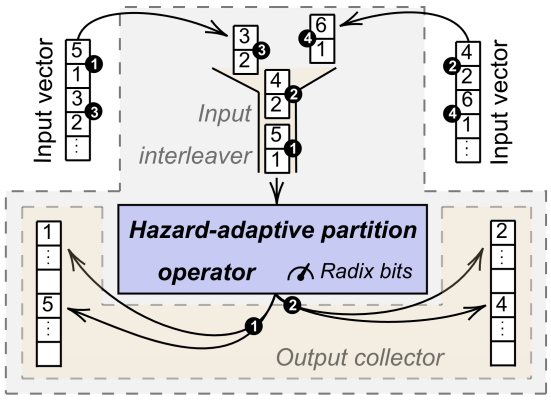


Fig. 5. Co-adaptive partition operator

During the data shuffling phase values are moved to their respective partition, each typically residing on a separate page, requiring a TLB entry [29]. If the fan-out exceeds the sTLB capacity and a random partition access pattern is required this causes sTLB store misses and subsequent data stalls. A high sTLB store miss rate is known as "TLB thrashing" [30].

The TLB footprint of the hazard-affected `PARTITION` algorithm is reduced by converting it to a multi-pass algorithm using fewer radix bits and recursively partitioning. By using a TLB footprint (fan-out) smaller than the sTLB capacity the algorithm will not suffer sTLB misses and is hazard-robust.

As the algorithm is hazard-robust the performance per tuple is broadly independent of the data. However, unlike `SELECT` there is a spectrum of hazard-affected algorithms, with varying fan-outs between the fully robust and single-pass algorithm. Utilizing this spectrum of algorithms maximizes the range of inputs the hazard-adaptive operator can process optimally.

Monitoring Heuristics: The dominant micro-architectural event for high fan-out radix partitioning is last-level sTLB store misses. Therefore, the system monitors the TLB store miss rate to adjust the hazard-adaptive operator's robustness. As the TLB store miss rate for a specific fan-out depends on the (order-dependent) key data characteristic of "clusteredness" (which determines the number of "cache-hot" partitions) it is a good proxy metric for clusteredness.

Dispatching Logic: The optimal number of radix bits to partition an input depends on the clusteredness of the data, with greater clusteredness meaning that a larger number of radix bits can be used without generating excessive data hazards. As for all hazard-adaptive operators, the dispatcher starts executing micro-batches using the most hazard-affected algorithm, i.e., the highest radix bits value. Since this is a co-adaptive operator the input interleaver will alternate between the two tables and the monitor will send signals to the dispatcher based on the hazards generated by both tables.

When the dispatcher is signaled to increase robustness it decrements the number of radix bits and signals the output transformer to merge the necessary partitions in both output tables in the output collector to match the partitions associated with the new radix bits value. The associated histogram values of both tables are also updated, and in a multi-threaded setup, the synchronizer ensures consistency across threads.

This approach limits the operator's use to homogeneous data (Section VII-C) since the radix bits value can only be decreased. The radix bits value could be reset for recursive partitioning to deal with heterogeneous data (since the order of the data is retained within partitions) or retained if the data is expected to be homogeneous. Either choice could leave performance on the table and the optimization of this decision is left for future work. Since we retain the radix bits value, if the minimum value is reached no more adaptivity is possible and the remaining input is processed.

Application to Joins: We apply our co-adaptive partition operator to radix hash joins, where the partitioning phase constitutes the majority of execution time, regardless of join type (e.g., left, inner) [29], [30]. Consequently, improvements to the partitioning performance lead to a comparable, albeit slightly lower, performance gain in the overall join operator.

C. Expanding the Set of Hazard-Adaptive Operators

As discussed in Sections III-A and III-B, different algorithms involve trade-offs, making hazard-adaptivity applicable to all but the most trivial operators. For instance, a co-adaptive `JOIN` operator could combine a hash join with a sort-merge join, favoring the latter for its predictable access patterns when the hash join working set size is too large. Similarly, co-adaptive `INTERSECT` and `UNION` operators can combine hash-based and hazard-robust sort-merge algorithms. Furthermore, a hazard-adaptive `SORT` operator could use a high radix-bit radix sort coupled with a hazard-robust merge sort.

VI. System Integration

To test the presented concepts we must integrate them into a system. To achieve this we used the BOSS system.

A. BOSS System

To use hazard-adaptive operators without developing a new system, we build on an existing execution engine. Leveraging the extensibility of composable data management systems (DMS) we focus on operators amenable to adaptivity. Thus we use "BOSS" [31], our next-generation composable DMS.

Velox and Apache Arrow: The means of extensibility in BOSS is an engine. We combine the operators into a hazard-adaptive engine and integrate it with Apache Arrow [32] (for storage) and Velox [33] (for execution) with minimal overhead. In this system, the engines are loosely coupled, allowing us to easily augment or replace functionality in Velox with that of the hazard-adaptive engine with virtually no changes to Velox.

Non-Adaptive Engine: We found Velox's `GATHER` operator to have some overhead, so to avoid modifying the Velox kernel directly, we created a standalone "non-adaptive engine" with a non-adaptive `GATHER` operator and `JOIN` operator (accepting indexed tuples from the upstream `PARTITION` operator). However, this is an implementation detail and, for simplicity, we discuss the Velox and non-adaptive engines as one.

B. Cross-Engine Vectorization

Query evaluations start with the Apache Arrow storage engine loading tables. Next, the hazard-adaptive engine performs

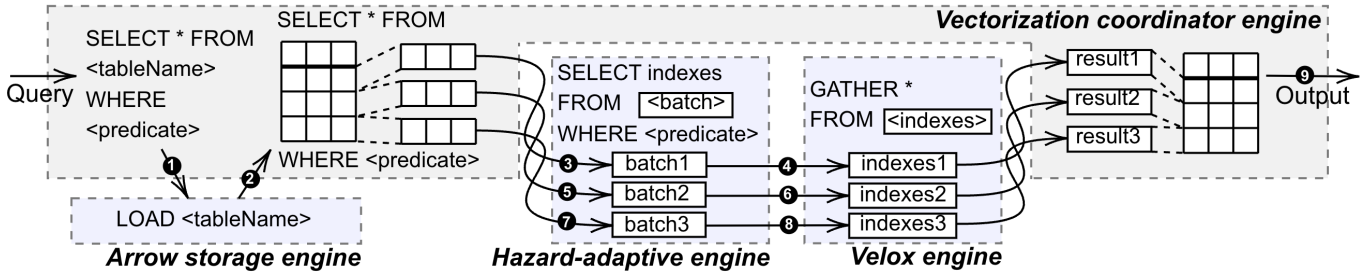


Fig. 6. Cross-engine vectorization in BOSS system pipeline

partial query evaluation on operators (or parts of operators) that can benefit from adaptivity, such as index creation for `SELECT` (Section IV-A). The remaining stages of these operators (e.g., gather following index generation in `SELECT`) and other unevaluated operators are handled by the Velox engine.

Velox is a vectorized execution engine. Therefore to achieve competitive performance the hazard-adaptive engine is vectorized (using ∼1MB vectors) in the style of X100 [34] or DuckDB [35]). However, BOSS does not impose an execution mode (i.e., it has no notion of vectorization) and would materialize all intermediate results. Therefore we create a "Vectorization coordinator engine" that splits queries into pipelineable chunks (vectors) and dispatches them to the relevant engines. Figure 6 shows this "cross-engine vectorization" architecture with numbers representing engine execution order. After the Arrow storage engine loads the tables ❶ they are split into vectors ❷, and passed through the pipeline of execution engines: the "Hazard-adaptive engine" and "Velox engine" (❸ to ❽). Finally, the results are unioned and returned ❾. Pipeline breakers must fully materialize their results and thus cannot be vectorized. Therefore, `GROUP` and `PARTITION` are batch-evaluated in the hazard-adaptive engine.

C. Parallelization

We leverage vectorization to perform parallelization across vectors since they can be processed independently. The individual results are collected and unioned within the vectorization coordinator engine. This makes parallelization straightforward, using thread local storage to store operator objects for reuse within each worker thread. However, since the operators are executed independently they are unaware of the DOP and thus the correct machine constant value to use within a heuristic. Therefore the vectorization coordinator engine also passes the DOP to the hazard-adaptive engine (embedded within the BOSS expression alongside the query).

D. Metaparameters and Trade-Offs

A key metric of adaptive algorithms is the maximum rate of change of the input data that the algorithm can adapt to. This metaparameter, which we call the "adaptive period", is a condition of adaptive algorithms and determines many implementation design choices. The input data determines the required adaptive period. Therefore, developing an adaptive algorithm with a static adaptive period to work across a range of datasets requires considering several trade-offs.

All of the switching mechanisms and tests discussed in Sections III-C and III-D 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 a satisfactory adaptive period (responsiveness) without generating too much overhead.

For our hazard-adaptive operators, we did not extensively tune the adaptive period, instead simply selecting a short adaptive period that maintained a low average overhead ($< 5\%$).

E. Performance Counters and Heuristics

Performance Counters: Performance counters can be read from within a container or VM [36] provided PMU access (Linux paranoid level ≤ 1) [37]. Additionally, since counters are per core, there will always be sufficient counters, as the active operator count never exceeds the core count.

Simplifying Heuristics: The performance of hazard-robust algorithms is not entirely constant, this is primarily due to two factors. Firstly, where the output size is variable it naturally affects performance. Secondly, multi-pass algorithms include step changes in performance depending on the number of passes. Therefore heuristics should account for these factors by adjusting the required frequency of events necessary for the monitor to observe before signaling the dispatcher to increase hazard robustness. For example, adjusting it based on the selectivity/cardinality of the data, or scaling it by the number of passes required for multi-pass algorithms (where this differs from the value used to create the heuristic). However, in practice, we found that a simple fixed value closely followed the optimal static implementation. This simplification is feasible due to the consistency of the tipping point across various inputs and the only minor impact of different output sizes.

Calibrating Machine Constants: Our experiments require ∼100 machine constants, however, as they are constant they can be computed at installation. An automated experimental binary-search calibration determines the switch-over point between foundational algorithms and measures the corresponding event frequency. During calibration, a low interference environment is preferable to capture maximum performance.

VII. Evaluation

We limit our evaluation to Intel CPUs. However, our approach and operators are equally applicable to AMD and ARM

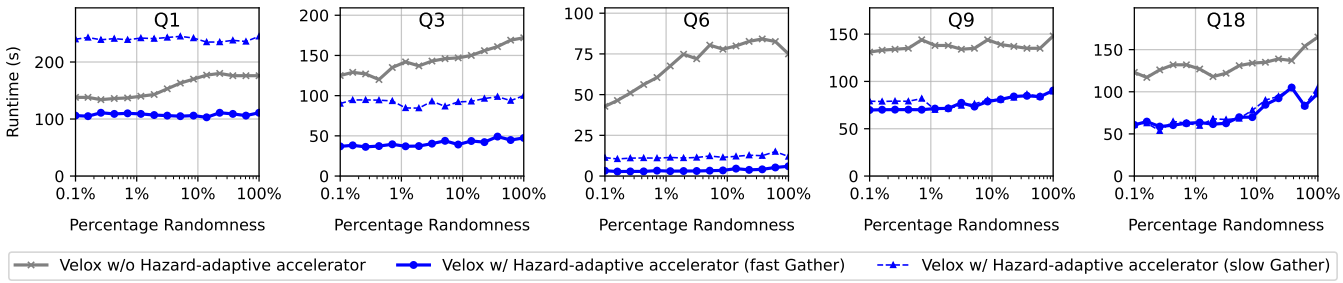


Fig. 7. Single-threaded runtime for TPC-H on five representation queries at SF100, varying sortedness of key tables

CPUs given their similar branch prediction, cache management, TLB usage, etc., and analogous performance counters).

A. Experimental Setup

Hardware: Experiments are performed on a server with two Intel Xeon Silver 4114 2.20 GHz CPUs, each with 10 physical cores, a 14 MB LLC cache, and 196 GB of memory. Each core has 1,536 TLB entries for 4KB pages (L2 TLB) and 4 entries for 1GB pages (L1 TLB). We limit multi-threaded experiments to a single NUMA node to exclude NUMA effects. We use Ubuntu 18.04 with Linux kernel 4.15.0-209 running in a Docker container and compile all code with Clang version 14 using flags `-O3 -march=native -std=c++20`.

Systems and Setup: To implement our system, we used BOSS v0.3, Apache Arrow v6.0.1, and Velox v0.0.1a0 (git rev. 164903523e39). Our implementation required two minor modifications to Velox: exposing the Task's internal child pools to take ownership of output data and adding support for arithmetic operators of mixed data types. When combined with our accelerator we merge Velox's output batches to achieve the desired size (i.e., 300K rows) in our code and leave Velox's parameters decoupled and as default. We measure execution time using Google Benchmark after three warm-up runs.

Workloads: For macro-benchmarks, we use TPC-H [38] (SF 100). Following de facto practice [39], we focus our evaluation on five queries that capture TPC-H's choke points [40].

For low cardinality string columns, the Arrow storage engine implements dictionary compression and updates the associated predicates. Due to the lack of support for non-dictionary compressed string columns in BOSS yet, Q9 is modified to filter the part table on p_retailprice rather than p_name and Q18 to group by o_orderkey only. However, the cardinalities are preserved. Q1, Q3, and Q6 are unchanged.

Query plans are hand-written using fixed heuristics: to decide the join order for each query, we iteratively pick the two smallest among all the pairs of relations that can be joined next and always use the smaller of the two on the build side.

Micro-benchmarks use 64-bit ints for keys/payloads.

B. Performance with TPC-H Benchmark

Of the five queries we run, Q1, Q3, and Q6 are dominated by Selection performance, Q9 by Join performance, and Q18 by Group performance. These collectively cover much of the problem space as most queries align with one of these patterns.

We first evaluate single-threaded performance on TPC-H. To test hazard-adaptive operator robustness (maximizing worst-case performance) we select dimensions to sort the TPC-H tables and progressively randomize them to create pathological cases for both hazard-affected and hazard-robust algorithms. This also replicates the full spectrum of data locality that is observed in real-world data as discussed in Section I.

We also examine performance on the default TPC-H tables and include multi-threaded performance (though the benefits of hazard-adaptive operators are orthogonal to parallelism).

Modified TPC-H Tables: To modify TPC-H tables, we vary at least one table on at least one column from fully sorted to fully random to maximize the performance differential of the dominant operators in the associated query. We believe our selected dimensions generate a near-minimum hazard frequency but cannot prove this. Therefore, other dimensions may create more pathological test cases, potentially increasing the maximum upside from the hazard-adaptive operators.

We include two accelerator results, one with the Velox Gather operator (slow Gather) which obscures some hazards due to overhead, and one with our own Gather operator (fast Gather). Implementing this in Velox would require reconfigurations (e.g., batch size), which we aimed to avoid, thus we keep it separate by leveraging our composable architecture.

Q1, Q3, and Q6 include large selections accounting for much of the query cost. We therefore sort the large tables (lineitem and orders) by the columns in the selection predicates (l_shipdate for Q1; l_shipdate and o_orderdate for Q3; and multi-level sorting by l_quantity, l_discount, and l_shipdate for Q6). Figure 7 shows Velox's performance degrades with increasing randomness due to branch mispredictions. This effect is worse for Q3 (selectivities of 48% and 54%) compared to Q1 (98%) as Q3's selectivies are closer to 50% increasing branch mispredictions exposure. Q6 performs worst given the first three predicate selectivities of 46%, 55%, 50%, and overall selectivity of only 1.9% making the remaining query cheap to evaluate. In contrast, the hazard-adaptive accelerator (with fast Gather) provides robust performance from sorted to random and performance benefits up to $\sim$20$\times$. Since selections are important in these queries, Gather performance is key and the overhead of the slow Gather is significant.

Joins dominate Q9's cost. The orders table (150M rows) join requires the largest build side ($\sim$30M rows) and therefore has the potential to generate the most hazards in a hash-join. Consequently, we sort lineitem and orders by orderKey. However, since the probe side key (orders) is the primary key (orderKey) it is unique and, as a result, the sortedness

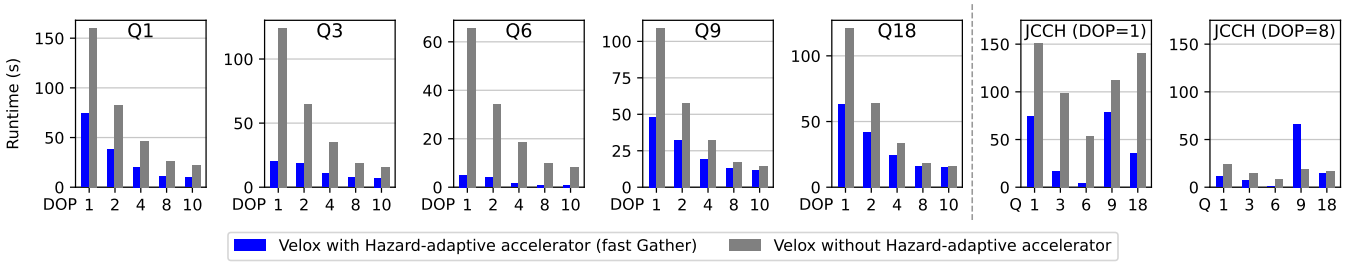


Fig. 8. Runtime for TPC-H and JCC-H on five representative queries at SF 100, varying degrees of parallelism (DOP)

has minimal impact on reducing hazards during the probe phase (a lower cardinality would mean repeated keys during probing and a greater cache hit frequency). We see this on Q9 in Figure 7 with the performance of Velox not degrading with increased randomness. This is because a high cache miss frequency is observed regardless of the degree of randomness. However, this is still a pathological case for the co-adaptive `PARTITION` operator with the TLB miss rate determined by the sortedness of the keys. By adapting to locality within both keys the co-adaptive partition operator delivers high performance and robustness across the problem space.

Q18 is dominated by the cost of the high cardinality (150M) aggregation of the lineitem table (600M rows) on orderKey and the large join between the customers (15M rows) and orders (150M rows) tables. Therefore we sort lineitem by orderKey, and orders and customers by custKey. Velox shows some performance degradation, however, this is minimized by significant prefetching during hash probing and overhead due to Velox's small internal batch size of 1024. Without prefetching and with a larger internal batch size (including joins) we observed a performance degradation of at least 2×. The hazard-adaptive accelerator improves performance, in part due to switching to sort-based aggregation at ∼50% randomness to prevent excessive cache misses.

Default TPC-H Tables: Figure 8 shows single and multi-threaded performance on default TPC-H tables. Even on unmodified data, the hazard-adaptive accelerator improves single-threaded performance by ∼2-20× by accurately selecting the optimal operator implementation, a task traditional optimizers struggle with due to the difficulty of capturing order-dependent metrics a priori (Section I). In this case, the adaptive operators adapt to unpredictable data in the selections and the sortedness of the data in the large joins/aggregations.

Velox demonstrates good scalability with increased DOP, achieving near-linear speedup. The hazard-adaptive accelerator also performs well with increased DOP, becoming memory-bound in Q6. Across all queries and DOP values, the hazard-adaptive accelerator provides Velox with a performance benefit, though this is reduced for Q9 and Q18, particularly at high DOP. This is due to the large joins, which require the hazard-adaptive `PARTITION` operator. Within this operator, efficiently parallelizing the first radix partitioning pass is a significant engineering challenge due to extensive thread result merging. In contrast, the overhead of synchronizing all threads to the same point in the adaptivity space is negligible.

JCC-H: Figure 8 also includes JCC-H [41], an alternative dataset for TPC-H which includes join crossing correlations (JCC). Performance is generally unchanged from TPC-H, except for the hazard-adaptive accelerated Q9 with DOP=8. This is due to using radix-partitioned hash joins, where skew is a known weakness since it causes to poor load balancing [30]. We leave this as an opportunity for future work.

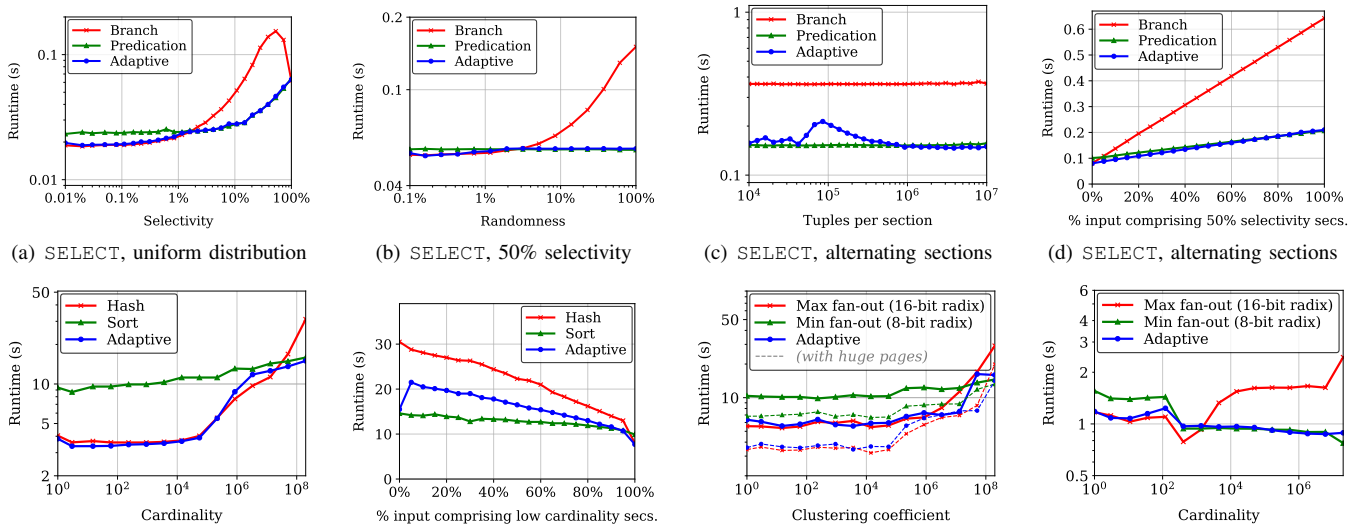
C. Micro-Benchmarks

SELECT: Figure 9 (a)-(d) contain the micro-benchmarks for `SELECT`. Figures (a) and (b) show operator performance on homogeneous datasets where the defining data characteristics remain consistent. Figure (a) varies the higher-level order-invariant characteristic of selectivity, while Figure (b) varies the order-dependent characteristic of randomness (i.e., % of randomly shuffled data in a sorted input) at a fixed selectivity of 50%. These figures cover the full spectrum for selectivity and randomness, representing all cases for the `SELECT` predicate. Both figures show the hazard-adaptive operator tracks the optimal static implementation with negligible overhead.

Figures (c) and (d) show operator performance on heterogeneous data, where data characteristics vary throughout the dataset. Both figures comprise sections of high (50%) and low (0%) selectivity. Figure (c) varies the length of each section (100M total tuples) to test inputs ranging from highly dynamic to relatively stable. The performance of the foundational algorithms is independent of the tuples per 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 foundational algorithms. This is because the hazard-adaptive operator selects the optimal foundational algorithm to process each section thereby achieving better performance than any static implementation.

Conversely, when there are a small number of tuples in each section the adaptive algorithm cannot work effectively as the input changes more frequently than the algorithm's adaptive period (50K tuples for the `SELECT` operator). The adaptive algorithm cannot adapt fast enough in such a dynamic scenario and will predominantly run the hazard-robust algorithm.

Between these extremes, performance decreases when section lengths are on the order of the adaptive period. This is because the input changes too fast for the adaptive algorithm to respond to, often causing selection of the incorrect implementation for the next micro-batch. This limitation of a static



(a) SELECT, uniform distribution (b) SELECT, 50% selectivity (c) SELECT, alternating sections (d) SELECT, alternating sections

(e) GROUP, uniform distribution (f) GROUP, alternating sections (g) PARTITION, 200M cardinality (h) PARTITION, uniform distribution

Fig. 9. Micro-benchmarks: Individual hazard-adaptive operator performance versus associated foundational algorithms

adaptive period can be remedied by using a dynamic heuristic for the adaptive period which we evaluate later in this section.

In Figure (d) we fix the length of input sections (above the adaptive period) and vary the number of 0% selectivity sections relative to 50% selectivity sections. When 100% of the input is 0% selectivity sections, it corresponds to a uniformly distributed input with 50% selectivity; at the other extreme, it corresponds to a 0% selectivity input. Between these 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 outperforms both foundational algorithms.

GROUP: Figures (e) and (f) contain the micro-benchmarks for GROUP. Figure (e) varies the cardinality of the input with uniform data (i.e., homogeneous data) and shows that the hazard-adaptive operator tracks the optimal static implementation with minimal overhead due to lazily initializing the hash table and repeatedly quickly identifying the optimal static implementation using the page fault rate.

Figure (f) shows operator performance on heterogeneous data, with sections varying between high and low cardinality. The hazard-adaptive operator tracks the optimal static implementation (visible at the extremes), however, it incurs greater overhead (observable as the "Adaptive" line above rather than tracking the "Sort" line). The switch to the sort-based algorithm is triggered by the last-level cache miss rate (rather than the page fault miss rate which can only identify the tipping point when the hash table is empty, i.e., homogeneous datasets) which requires processing more tuples to get a converged and statistically accurate hazard rate. This increases the overhead due to more cache misses and, more importantly, page faults (requiring page initialization). This is a key challenge of a hazard-adaptive GROUP operator that can optimally process both homogeneous and heterogeneous data.

Different aggregate operators (e.g., sum, average) in Figures (e) and (f) yielded no observable performance difference.

JOIN (PARTITION): Figures (g) and (h) show the micro-benchmarks for PARTITION, both on homogeneous data since this operator is not suited to heterogeneous data as described in Section V-B. Figure (g) varies the order-dependent key data characteristics of clustering, whereas Figure (h) varies the order-invariant data characteristic of cardinality.

The hazard-adaptive operator tracks the optimal static implementation with only minor overhead in both figures. For a clustering coefficient[6] of $\sim 10^6 - 10^7$ in Figure (g) its performance exceeds that of both static implementations. This is because the hazard-adaptive PARTITION operator spans a spectrum of hazard-affected algorithms (one per radix bits value). For some clustering values, an intermediate radix value processes the input optimally. Compared to all the static implementations with radix bit values between 8-16 the hazard-adaptive operator closely follows the optimal static implementation. This illustrates the benefit of combining multiple hazard-affected foundational algorithms. We found this phenomenon to be far more pronounced on other machines.

Figure (g) also includes the results when forcing the OS to use huge pages, illustrated by the dashed lines. This machine contains 4 entries for huge pages, all in the L1 TLB. This greater use of the L1 TLB versus L2 TLB means that all the algorithms run faster, and the point at which TLB thrashing occurs is delayed due to the large working set size associated with the TLB capacity. However, at the point of TLB thrashing the process of adaptivity works the same as for normal pages, observed at the far right-hand side of the figure.

In Figure (h) values range from 1 to a fixed maximum value of 20M. For low cardinality datasets, the minimum fan-out algorithm requires an extra partitioning pass due to many repeated small values (partitioning starting from the most significant bit). This results in a "step-shaped" performance improvement at a cardinality of $\sim 10^2$ when one partitioning pass is sufficient. Near this point, the maximum fan-out

[6]Defined in footnote (4).

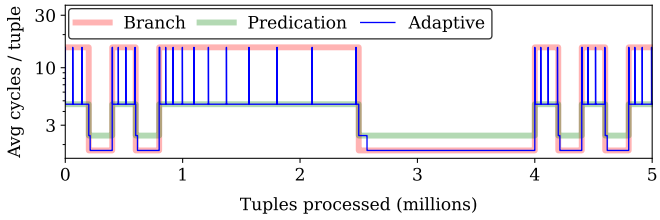


Fig. 10. Dynamic heuristic for SELECT on dynamic data

algorithm begins to suffer from TLB thrashing.

Dynamic Heuristic: To use counter readings instead of runtime exploration, the hazard-adaptive switching heuristic requires a static component that quantifies the relative performance of the foundational algorithms a priori (Section III-C). This is achieved through a one-time automated calibration that determines threshold hazard frequencies (Section VI-E). However, to minimize runtime overhead, the frequency of counter readings and algorithm probing can be dynamically adjusted.

Figure 10 presents an experiment using a dynamic heuristic for SELECT that adjusts the number of tuples between tests (adaptive period) to the rate of change of the data locality. The input varies between switching quickly and slowly between high and low predictability. Initially, probes (these tests appear as vertical lines) are frequent, but the interval increases geometrically until the robustness is changed, at which point the interval resets. As shown in Figure 10, this heuristic enables effective adaptivity in highly dynamic scenarios while minimizing overhead by reducing test frequency when possible.

Overhead: Across all experiments, reading counters and probing alternative algorithms incur a maximum overhead of ∼15%. However, as Figures 9 (a, b, e, g, h) show, the overhead on homogeneous datasets versus the non-adaptive equivalent is generally minimal. Note that the overhead in Figure 9 (f) is due to the non-linear hash table initialization cost, not adaptivity.

Parallelization: When the tipping point depends on a shared resource, parallelization alters resource contention and shifts the tipping point. For SELECT, predication requires more memory bandwidth, pushing the tipping point to favor the branch-based implementation at higher unpredictability with increased DOP. For GROUP, increased shared cache contention shifts the tipping point toward the sort-based implementation at lower variability. Conversely, the foundational algorithm for PARTITION is independent of shared resources (each core has its own TLB), and is unaffected by parallelization.

D. Versus Black-Box Micro-Adaptivity

White-box hazard-adaptivity differs fundamentally from black-box micro-adaptivity by extending applicability to a broader range of operators (e.g., pipeline breakers that require internal state access) and to datasets with varying data locality (black-box methods require a mechanism to detect locality changes and restart optimization). While the two approaches are not directly comparable, overhead comparisons versus an oracle show that black-box test-and-learn frameworks incur a minimum ∼15% overhead [8] due to the learning phase, whereas hazard-adaptive operators achieve minimal overhead.

VIII. Related Work

Data characteristics are traditionally captured in histograms [42], however, these fail to capture patterns like auto-correlation. Pre-execution methods to determine order-dependent metrics are complex and brittle [5]–[7]. Our approach eliminates pre-execution analysis, using micro-adaptivity to select the physical operator instead. Micro-adaptivity was introduced by Răducanu et al. [8], where they create multiple operator implementations via techniques such as loop fission/unrolling and using different compilers. A multi-armed bandit algorithm [43], [44] then optimizes performance through "exploration" and "exploitation" phases [8]. Kaftan et al. [45] expand this approach, but highlight challenges in quickly and accurately selecting the optimal operator implementation, with overheads of at least ∼15% [8] and scaling proportional to the number of implementations [45].

Grizzly [9], a stream processing engine, uses counters for two "adaptive optimizations". Fine-grained code instrumentation identifies optimizations and counters trigger re-optimizations in two areas. Firstly, identifying changes in predicate selectivity via branch mispredictions, triggering a profiling phase that incurs a ∼50% drop in throughput. Secondly, determining the use of a shared versus independent hash table in a multi-threaded implementation using simultaneous exclusive accesses to cache lines to detect data skew.

Zeuch et al. [15] use counters without code instrumentation to drive progressive optimization [46] at the overall query level by triggering a re-optimization process based on runtime comparisons of counter readings against cost models [15]. However, using counters was only explored to measure selection output cardinalities to optimize the order of selection predicates of a single table. Similarly to Zeuch et al., we aim to maximize the potential of counters, in our case applied to the paradigm of micro-adaptivity. We also make our approach widely applicable by creating a generalized architecture.

IX. Conclusion

Micro-architectural hazards are a key challenge for in-memory data processing systems. However, their prevalence depends on patterns within the data, making static query optimization ill-suited for this problem. We propose "white-box" micro-adaptive operators that continuously monitor micro-architectural hazards and advise switches to alternative implementations. We present architectures for hazard-adaptive and co-adaptive operators, implement key relational operators, and develop an accelerator engine for Velox based on a next-generation composable data management system.

We show that white-box micro-adaptivity accelerates Velox up to 20× on TPC-H benchmarks and is robust to varying degrees of data locality, including pathological cases. Micro-benchmarks confirm that hazard-adaptive operators match the optimal static operator performance with minimal overhead.

Hazard-adaptivity is a fundamentally different approach to prior work on micro-adaptivity, extending the paradigm to more complex operators (e.g., aggregations) and datasets with varying locality, whilst simultaneously minimizing overhead.

REFERENCES

[1] K. A. Ross, "Selection conditions in main memory," *ACM Trans. Database Syst.*, vol. 29, no. 1, p. 132–161, mar 2004. [Online]. Available: https://doi.org/10.1145/974750.974755

[2] IMDb, Inc., "Imdb non-commercial datasets," 2024. [Online]. Available: https://developer.imdb.com/non-commercial-datasets/

[3] International Monetary Fund (IMF), "Imf climate change indicators dashboard," 2024. [Online]. Available: https://climatedata.imf.org/pages/climatechange-data

[4] S. Naskar and J. P. Naveen. (2020) Histograms based on varying data distribution. Hewlett Packard Enterprise Development LP. [Online]. Available: https://patents.google.com/patent/US10628442B1/en

[5] S. Ben-Moshe, Y. Kanza, E. Fischer, A. Matsliah, M. Fischer, and C. Staelin, "Detecting and exploiting near-sortedness for efficient relational query evaluation," in *Proceedings of the 14th International Conference on Database Theory*, ser. ICDT '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 256–267. [Online]. Available: https://doi.org/10.1145/1938551.1938584

[6] P. Gopalan, T. S. Jayram, R. Krauthgamer, and R. Kumar, "Estimating the sortedness of a data stream," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 318–327.

[7] M. Ajtai, T. S. Jayram, R. Kumar, and D. Sivakumar, "Approximate counting of inversions in a data stream," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 370–379. [Online]. Available: https://doi.org/10.1145/509907.509964

[8] B. Răducanu, P. Boncz, and M. Zukowski, "Micro adaptivity in vectorwise," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1231–1242. [Online]. Available: https://doi.org/10.1145/2463676.2465292

[9] P. M. Grulich, B. Sebastian, S. Zeuch, J. Traub, J. v. Bleichert, Z. Chen, T. Rabl, and V. Markl, "Grizzly: Efficient stream processing through adaptive query compilation," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2487–2503. [Online]. Available: https://doi.org/10.1145/3318464.3389739

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[11] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2008, pp. 59–67.

[12] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, ser. STC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 71–76. [Online]. Available: https://doi.org/10.1145/2046582.2046596

[13] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?" in *2008 IEEE International Symposium on Workload Characterization*, 2008, pp. 141–150.

[14] S. Eranian, "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)*, ser. MSPC '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 26–30. [Online]. Available: https://doi.org/10.1145/1353522.1353531

[15] S. Zeuch, H. Pirk, and J.-C. Freytag, "Non-invasive progressive optimization for in-memory databases," *Proc. VLDB Endow.*, vol. 9, no. 14, p. 1659–1670, oct 2016. [Online]. Available: https://doi.org/10.14778/3007328.3007332

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

[17] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: Association for Computing Machinery, 1994, p. 196–206. [Online]. Available: https://doi.org/10.1145/192724.192753

[18] A. Crotty, A. Galakatos, and T. Kraska, "Getting swole: Generating access-aware code with predicate pullups," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1273–1284.

[19] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, no. 2007, p. 2007, 2007.

[20] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl, "Generating custom code for efficient query execution on heterogeneous processors," *The VLDB Journal*, vol. 27, pp. 797–822, 2018.

[21] S. I. F. G. N. Nes and S. M. S. M. M. Kersten, "Monetdb: Two decades of research in column-oriented database architectures," *Data Engineering*, vol. 40, 2012.

[22] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking simd vectorization for in-memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1493–1508. [Online]. Available: https://doi.org/10.1145/2723372.2747645

[23] S. McFarling and J. Hennesey, "Reducing the cost of branches," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Washington, DC, USA: IEEE Computer Society Press, 1986, p. 396–403.

[24] Tessil, "Tessil/robin-map," 2023, mIT License. [Online]. Available: https://github.com/Tessil/robin-map

[25] P. Celis, P.-A. Larson, and J. I. Munro, "Robin hood hashing," in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, 1985, pp. 281–288.

[26] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," *J. ACM*, vol. 65, no. 3, mar 2018. [Online]. Available: https://doi.org/10.1145/3180143

[27] S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *IEEE Trans. on Knowl. and Data Eng.*, vol. 14, no. 4, p. 709–730, jul 2002. [Online]. Available: https://doi.org/10.1109/TKDE.2002.1019210

[28] Z. Zhang, H. Deshmukh, and J. M. Patel, "Data partitioning for in-memory systems: Myths, challenges, and opportunities." in *CIDR*, 2019.

[29] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 362–373.

[30] M. Bandle, J. Giceva, and T. Neumann, "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. [Online]. Available: https://api.semanticscholar.org/CorpusID:231978821

[31] H. Mohr-Daurat, X. Sun, and H. Pirk, "Boss - an architecture for database kernel composition," *Proc. VLDB Endow.*, vol. 17, no. 4, p. 877–890, mar 2024. [Online]. Available: https://doi.org/10.14778/3636218.3636239

[32] Apache Software Foundation, "Apache arrow," 2023. [Online]. Available: https://arrow.apache.org/

[33] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. Sakka, K. Pai, W. He, and B. Chattopadhyay, "Velox: meta's unified execution engine," *Proc. VLDB Endow.*, vol. 15, no. 12, p. 3372–3384, Aug. 2022. [Online]. Available: https://doi.org/10.14778/3554821.3554829

[34] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman, "Monetdb/x100 - a dbms in the cpu cache," *IEEE Data Eng. Bull.*, vol. 28, pp. 17–22, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:18052553

[35] M. Raasveldt and H. Muehleisen, "Duckdb." [Online]. Available: https://github.com/duckdb/duckdb

[36] J. Du, N. Sehrawat, and W. Zwaenepoel, "Performance profiling of virtual machines," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 3–14. [Online]. Available: https://doi.org/10.1145/1952682.1952686

[37] T. U. of Tennessee, "Papi: Performance application programming interface," Innovative Computing Laboratory, 2023. [Online]. Available: http://icl.utk.edu/papi/

[38] Transaction Processing Performance Council, "TPC-H Benchmark," https://www.tpc.org/tpch/, 2024, accessed: 2024-05-22.

[39] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *Proc. VLDB Endow.*, vol. 11, no. 13, p. 2209–2222, sep 2018. [Online]. Available: https://doi.org/10.14778/3275366.3284966

[40] P. Boncz, T. Neumann, and O. Erling, "Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark," in *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*. Berlin, Heidelberg: Springer-Verlag, 2013, p. 61–76. [Online]. Available: https://doi.org/10.1007/978-3-319-04936-6_5

[41] P. Boncz, A.-C. Anatiotis, and S. Kläbe, "Jcc-h: Adding join crossing correlations with skew to tpc-h," in *Performance Evaluation and Benchmarking for the Analytics Era*, no. 10661, Aug. 2017, pp. 103–119.

[42] S. Chaudhuri, R. Motwani, and V. Narasayya, "Random sampling for histogram construction: How much is enough?" in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 436–447. [Online]. Available: https://doi.org/10.1145/276304.276343

[43] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, pp. 235–256, 2002.

[44] S. Bubeck, N. Cesa-Bianchi *et al.*, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Foundations and Trends® in Machine Learning*, vol. 5, no. 1, pp. 1–122, 2012.

[45] T. Kaftan, M. Balazinska, A. Cheung, and J. Gehrke, "Cuttlefish: A lightweight primitive for adaptive query processing," 2018.

[46] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 659–670. [Online]. Available: https://doi.org/10.1145/1007568.1007642