POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance

David Justen*
TU Berlin

Daniel Ritter
SAP

Campbell Fraser
Google

Andrew Lamb
Nga Tran
InfluxData

Allison Lee
Snowflake

Thomas Bodner
Hasso Plattner Institute
University of Potsdam

Mhd Yamen Haddad
INRIA, Ecole Polytechnique

Steffen Zeuch
Volker Markl
TU Berlin

Matthias Boehm
TU Berlin

ABSTRACT

Join ordering and query optimization are crucial for query performance but remain challenging due to unknown or changing characteristics of query intermediates, especially for complex queries with many joins. Over the past two decades, a spectrum of techniques for adaptive query processing (AQP)—including inter-/intra-operator adaptivity and tuple routing—have been proposed to address these challenges. However, commercial database systems in practice do not implement holistic AQP techniques because they increase the system complexity (e.g., intertwined planning and execution) and thus, complicate debugging and testing. Additionally, existing approaches may incur large overheads, leading to problematic performance regressions. In this paper, we introduce POLAR, a simple yet very effective technique for a self-regulating selection of alternative join orderings with bounded overhead. We enhance left-deep join pipelines with alternative join orders, perform regret-bounded tuple routing to find and validate "plans of least resistance", and then process the majority of tuple batches through these plans. We study different join order selection techniques, different routing strategies, and a variety of workload characteristics. Our experiments with a POLAR prototype in DuckDB show runtime improvements of up to 9x and less than 7% overhead for all benchmark queries, while outperforming state-of-the-art AQP systems by up to 15x.

PVLDB Reference Format:
David Justen, Daniel Ritter, Campbell Fraser, Andrew Lamb, Nga Tran, Allison Lee, Thomas Bodner, Mhd Yamen Haddad, Steffen Zeuch, Volker Markl, and Matthias Boehm. POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance. PVLDB, 17(6): 1350 - 1363, 2024.
doi:10.14778/3648160.3648175

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/damslab/reproducibility/tree/master/vldb2024-POLAR.

*Corresponding author's email: david.justen@tu-berlin.de

Proceedings of the VLDB Endowment, Vol. 17, No. 6 ISSN 2150-8097.
doi:10.14778/3648160.3648175

1 INTRODUCTION

Cost-based query optimizations [72, 87] for selecting optimal join orders, join methods, and data access paths are crucial for the end-to-end performance of analytical queries. State-of-the-art exact join ordering algorithms such as DPsize [42, 87], DPsub, DPcpp [73], and DPhyp [74] rely on dynamic programming for efficient enumeration. These algorithms yield optimal execution plans, but only under the assumption of precise cardinality estimates.

Cardinality Estimation Challenges: Estimating precise cardinalities for intermediate results of complex queries remains a stubbornly difficult problem [65]. First, most systems assume uniform distributions (no skew) and independence of predicates (no correlation) [50]. These simplifying assumptions often cause underestimation, which is problematic due to plan choices with poor asymptotic behavior (e. g., nested-loop joins), which perform poorly for larger intermediates [50, 65]. Second, too coarse-grained statistics (e. g., histograms [58] or sketches [55]) may misrepresent clustered data. Third, user-defined functions and new environments (e. g., federated, raw data) often do not allow obtaining statistics [48, 56, 84]. Fourth, complex queries with many operators are difficult to estimate because errors propagate exponentially [52, 55, 76]. Although recent work on ML-based estimators [35, 60, 98], learning to distrust certain estimates [70], and learning to rank plans [14] offer benefits, they do not solve all problems above.

Adaptive Query Processing (AQP): In the past two decades, a spectrum of AQP techniques [10, 29, 30, 53] has been devised to address the challenges of unknown and changing data characteristics. Many AQP techniques follow the classical MAPE control loop of monitoring, analyzing, planning, and executing [4, 49, 53]. Existing techniques include inter-query optimization with learned cardinalities for expressions [23, 25, 89], late binding with re-optimization at pipeline breakers [29] or parameter binding [17], inter-operator re-optimization at checkpoints [57], progressive and pro-active re-optimization with validity ranges [11, 71], intra-operator adaptivity with union stitch-up plans [54], intra-query adaptivity via reinforcement learning [93, 94, 97], as well as tuple routing policies in Eddies [8, 9, 16, 28]. Many of these strategies require both optimizer and runtime extensions for effective and efficient adaptivity.

Robust Query Processing: An alternative mitigation strategy for poor cardinality estimates is robust query processing [44]. The influential Picasso project [43] on plan diagrams [83] revealed that

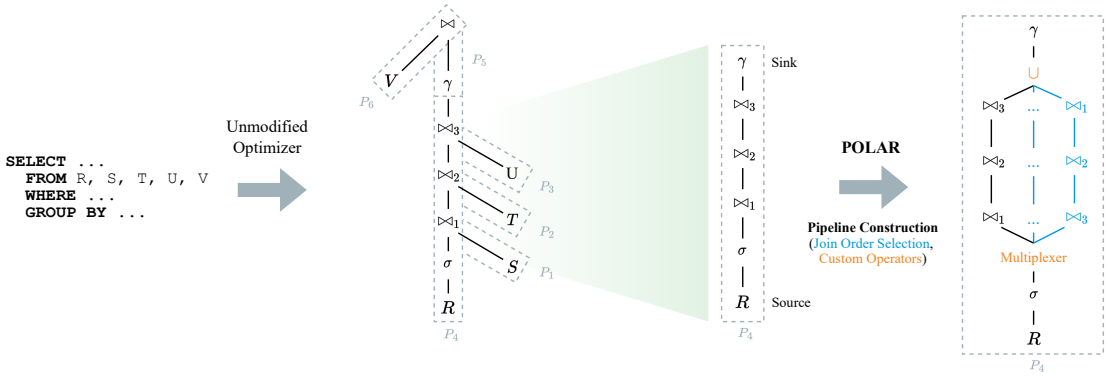

Figure 1: POLAR pipeline compilation from input query over standard, pipelined plan to POLAR pipeline.

state-of-the-art DBMS compile many very specialized plans that are only optimal in a small subspace of cardinalities. Since these cardinalities are difficult to estimate, robust query processing seeks to find a small number of plans that together cover a broad range of cardinalities [32, 33]. Despite a sequence of valuable improvements [3, 33, 34, 38] (including so-called plan bouquets [34]), many of these strategies are offline approaches and largely intractable for intra-query or intra-operator adaptation during runtime [44].

POLAR Overview: Although AQP comprises many valuable ideas, only very few are implemented by data(base) systems in practice. We attribute this largely to the induced complexity of intertwining planning and execution, difficulties in testing and debugging, and potential performance regressions due to overheads of adaptivity. Inspired by tuple-routing and self-scheduling (queue-based) systems, we introduce POLAR as a novel adaptive processing approach of join pipelines. We enhance left-deep join pipelines with alternative join orders during planning, perform regret-bounded tuple routing for exploration, and process most data through *plans of least resistance* (i.e., plans with few intermediates). In contrast to tuple routing in Eddies and SkinnerDB, POLAR is non-invasive to the optimizer and runtime, has low and bounded overhead, and does not require state management (e.g., partially-built hash tables).

Contributions: Our primary contribution is the concept of plans of least resistance (POLAR) as a new AQP technique designed for simple system integration and bounded overhead. Our detailed contributions include the novel POLAR design and its evaluation:

- *Pipeline Design:* We introduce a holistic, non-invasive pipeline design from objectives over pipeline compilation and join order selection to parallel pipeline processing with performance tracking (Section 2).
- *Routing Strategies:* We propose an extensible multiplexer operator and several routing strategies, as well as describe their trade-offs and runtime characteristics (Section 3).
- *SSB-skew Benchmark:* As a basis for evaluating AQP systems on data with correlations and clustering, we introduce and share the new SSB-skew benchmark (SSB-skew repository).
- *Experiments:* Using a variety of benchmarks (JOB, SSB, SSB-skew), we systematically study the performance characteristics of a POLAR prototype in DuckDB [81]. We evaluate different join order selection and routing strategies and compare with different AQP systems (Section 5).

2 PIPELINE DESIGN

POLAR is an adaptive join processing approach designed for non-invasive integration into common database systems with support for vectorization and operator pipelining. In contrast to other AQP techniques, POLAR pipelines do not require fine-grained intertwining of existing optimizers and runtime systems. As shown in Figure 1 (right), we enhance amenable pipelines with additional join orders. At runtime, we measure the performance of these orders and route tuples to well-performing orders while exploring others using a regret budget. This section introduces POLAR's design objectives and gives an overview of the compilation and execution of POLAR pipelines and related essential primitives.

2.1 Design Objectives

We largely attribute the stagnant adoption of adaptive join processing approaches to difficult system integration and problematic performance regressions. Accordingly, we propose a non-invasive approach specifically designed for low and bounded overhead. In the following, we define these objectives and distinguish POLAR from existing AQP systems along these dimensions (in Table 1). As bitmap filtering is increasingly adopted in popular database systems [31, 63], we also compare POLAR to Lookahead Information Passing [102] (LIP) as a representative candidate.

Non-invasive System Integration: We call an AQP system non-invasive if it clearly separates compilation from execution (no plan generation at runtime) because this separation simplifies debugging and testing. Furthermore, for simple system integration and maintenance, a non-invasive AQP system should reuse existing infrastructure such as the query optimizer and physical operators. Eddies [9] and SkinnerDB [93] intertwine query processing phases, and require specific optimizers and operators. LIP mostly adheres to non-invasiveness but generates new bloom filter orders at runtime.

Table 1: System Integration and Bounded Overhead.

Approach	Separation	Reuse	Bounded Overhead
Eddies	×	×	×
SkinnerDB	×	×	✓
LIP	(✓)	✓	×
POLAR	✓	✓	(✓)

Bounded Overhead: Preventing major performance regressions on any workload—even if adaptation is not beneficial—is essential for enabling AQP by default. Accordingly, an AQP system should provide bounds for the overhead of adaptivity (e.g., exploration of alternative join orders). Eddies and LIP do not define bounds for their overheads, SkinnerDB gives a strict bound based on the single best (robust) join plan, and POLAR provides a probabilistic bound on the number of intermediates incurred for exploration.

2.2 POLAR Pipeline Compilation

During query compilation (cf. Figure 1), we replace amenable operator pipelines with POLAR pipelines. A POLAR pipeline contains alternative join orders and two dedicated operators: a *multiplexer* (for tuple routing) and an *adaptive union* $\cup$ (for result consolidation). In the following, we describe the pipeline selection, the dedicated operators, and the pipeline transformation in more detail.

Pipeline Construction: After query optimization and generation of a query execution plan—consisting of multiple operator pipelines—POLAR replaces all pipelines of left-deep join trees with at least two joins (where the right-hand-sides are build sides, and left-hand-side intermediates are probed in a pipelined fashion). The pipeline's source is the left-most node and fixed as POLAR's source of input tuples for the tuple routing. Our approach generates a set of alternative join orders using a *join order selection* algorithm (cf. Section 2.3) and includes them with the original join order. Although focusing only on existing join pipelines is limiting, it allows for a system integration without changing the query optimizer and robust performance that is at least as good as the original plan. If the original plan contains pipelines with multiple joins, POLAR can improve these pipelines via alternative orders.

Custom Operators: Additionally, we introduce two new operators into each POLAR pipeline for tuple routing. The *multiplexer* (cf. Section 3) accepts bags of tuples from the source table and determines the next path and the number of tuples to route to that path. It uses performance indicators from previous path runs to make routing decisions. After each path run, a lightweight *adaptive union* operator processes the results from the last join. Besides normal union-all semantics, this operator re-arranges the additional columns from the joins to a consistent order of the original plan.

Pipeline Transformation: Finally, we replace the existing operator pipeline with a POLAR pipeline consisting of the multiplexer, the set of join orders, and the adaptive union. The individual join orders reuse the read-only hash tables of the build sides without redundant allocation. This transformation does not break the pipeline to prevent further materialization and multiplexing. Other non-blocking operators (e. g., projections, filters) and the pipeline sink (e. g., aggregation) also remain unchanged. The POLAR pipeline allocates space for state per alternative join order (e.g., operator caches, intermediate result vectors). Thus, the space overhead grows linearly with the number of join orders but is negligible in practice.

2.3 Join Order Selection

When generating alternative join orders for individual pipelines, we aim to compose a diverse set of orders that could handle a wide range of mis-estimated cardinalities and clustered data. The selection strategy should find good plans early (i. e., be robust for large pipelines), should not re-invoke the query optimizer (i. e., ensure non-invasive integration and low overhead), and should pick few complementary join orders (i. e., reduce the amount of exploration overhead). To this end, we propose two novel, anytime join order selection strategies that do not require optimizer invocations, as well as additional heuristics to serve as baselines.

Algorithm 1 Selectivity Space Sampling

Input: Joins J, Max Join Orders MAX, Max Iterations $maxi$
Output: Join Orders T

$T \leftarrow \emptyset, i \leftarrow 0$
$\mathbf{D} \leftarrow$ DISCRETIZESELECTIVITIES(J) // *exponential decay*
while $|T| < MAX \wedge i < maxi$ **do**
  $d_i \leftarrow$ SAMPLE$(\mathbf{D}), i \leftarrow i + 1$ // *$|J|$ selectivities*
  $T \leftarrow T \cup$ DPSIZE(J, d_i) // *keep distinct optimal join orders*
return T

Algorithm 2 Next-best Join Order Search

Input: Joins J, Edge Counts $n_0, \ldots, n_{|J|} \in N$, Max Join Orders MAX
Output: Join Orders T

$R \leftarrow \emptyset$
$S \leftarrow$ LEGALJOINCANDIDATES(R, J) // *Find first join candidates*
for $i \leftarrow 0$ **to** n_0 **do** // *Consider n_0 of the candidates*
  $s_i \leftarrow$ GETNEXTJOIN(S)
  $prio \leftarrow (|R|, i)$
  $pqueue$.ENQUEUE$(prio, R, s_i)$
  $S \leftarrow S \setminus s_i$
while !$pqueue$.EMPTY() and $|J| < MAX$ **do**
  $R, s \leftarrow pqueue$.DEQUEUE() // *Remove next item*
  $R \leftarrow R \oplus s$
  if $|R| = |J| - 1$ **then**
    $R \leftarrow R \oplus J \setminus R$ // *Append remaining join*
    $T \leftarrow T \cup R$
  else
    $S \leftarrow$ LEGALJOINCANDIDATES(R, J)
    for $i \leftarrow 0$ **to** $n_{|R|}$ **do**
      $s_i \leftarrow$ GETNEXTJOIN(S)
      $prio \leftarrow (|R|, i)$
      $pqueue$.ENQUEUE$(prio, R, s_i)$
      $S \leftarrow S \setminus s_i$
return $T \cup$ GETORIGINALJOINORDER()

S1 SELSAMPLING: For good coverage of complementary join orders, we propose selectivity-space sampling in Algorithm 1, a method to sample the space of unknown selectivities and generate optimal join orders for each sample. Inspired by progressive parametric query optimization [17], the discretized selectivities of base table predicates and joins form a d-dimensional grid, which we sample uniformly. For each sampled point, we then call DPsize with C_{out} cost model to generate the optimal plan for this scenario. In order to efficiently explore alternative orders, we use an exponential discretization (e.g., 0.1, 0.01, 0.001). For predicates and foreign key/primary key joins, we multiply this selectivity with $|R|$; for general joins with $|R \times S|$. Compared to brute-force creation of all left-deep join orders (2^{n-1} for chain, $n!$ for clique queries) [72], SELSAMPLING generates only up to MAX join orders, and can be terminated anytime (e.g., if $maxi$ samples were taken).

S2 NEXTBESTSEARCH: With Algorithm 2, we introduce an alternative join order selection approach called *next-best search* that

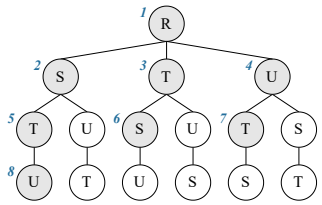

Figure 2: The Join Order Search Space as a Tree: A path from root to leaf denotes a sequence of joins. The visited nodes are marked in gray, and the digits indicate the order in which they are visited using the Next-Best Join Order Search. At step 8, the algorithm emits the first join order $\langle R, S, T, U \rangle$.

uses a priority queue to explore join candidates in a breadth-first search manner. The algorithm considers the search space a tree with parent-child sequences representing join orders. In this tree, the set of all paths from root to leaf node is the set of all legal join orders. Figure 2 shows an example of such a join order search tree. We explore the tree by retrieving all possible join candidates S for a given node R. The function GetNextJoin(S) then determines the $n_{|R|}$ fittest candidates, with $n_{|R|}$ being the number of edges to consider at a certain tree level. For each candidate s_i, the algorithm pushes an item containing the current node, the join candidate s_i, the tree level (i. e., the length of the current join sequence R), and the fitness index i. The priority queue compares its items based on the tree level and fitness index—similar to recent heuristic search strategies in join enumeration to maximize pruning [41]—so that the fittest join candidate in the lowest level is always the next item to process. Depending on the values for $N = (n_0, \ldots, n_{|J|})$, the maximum number of join orders MAX, and GetNextJoin, the algorithm allows finding different join order subsets. For example, setting N to increasing values favors diversity of join candidates for the rear joins of a join order. We utilize two versions of GetNextJoin:

- **S2a GetRandom:** Pick a random join candidate.
- **S2b GetMinCard:** Pick the join candidate with the lowest estimated cardinality.

Basic Heuristics: Besides the search-based approaches, we use two simple additional heuristics to generate alternative join orders:

- **S3 PushDown:** Permute the original join order such that each join gets pushed to be the first in the join order once if legal (assuming that the first join in the pipeline often has the largest performance impact [86]).
- **S4 PullUp:** Pull each join to the last position in the join order if possible (assuming that join blowups may be mitigated if the problematic join is pulled up to the end of the join order as other joins may filter its input).

2.4 Pipeline Execution

During query execution, POLAR routes tuples from the source table of a pipeline through multiple join paths. The pipeline executor of these paths reports the performance and calculates their *resistance* for future routing decisions. By using thread-local state (e.g., for tuple buffers and multiplexer state), POLAR can execute its pipelines in parallel without blocking. In this section, we discuss the related techniques for pipeline orchestration, our resistance metric, and parallel execution strategies in detail.

Pipeline Orchestration: To process a POLAR pipeline, the database system spawns a custom POLAR pipeline executor responsible for passing tuples to the operators according to the multiplexer's routing decisions. The executor fetches chunks (i. e., mini-batches or sets of tuples) from the source and passes them sequentially through the pipeline. The multiplexer consumes a chunk and returns an output chunk containing a subset of the input (or the whole input) and the index of the next join order to pass the subset to. If the multiplexer does not return all tuples from the input, the executor re-invokes the multiplexer with the same input chunk in the next iteration to emit the next tuple subset instead of fetching a new chunk from the source. After routing the chunk to its dedicated join order, the executor passes the chunk to the adaptive union, other pipeline operators, and finally the pipeline sink. During this process, the executor counts the number of intermediates appearing within the path as a performance indicator. We chose intermediates over time because they allow isolated observations (irrespective of other operators and parallelism), and the related C_{out} cost model is known to be simple yet accurate [65, 72]. After fully processing one multiplexer output chunk in a join order, the executor reports back the number of intermediates from that routing iteration to the multiplexer. This design allows adapting the *plans of least resistance* to clustered data. For example, table R from Figure 1 may have many matches with S and few with T for its first half of rows but behave the opposite for its second half. For this reason, POLAR never discards previously generated join orders.

Resistance Metric: As a proxy for performance, the POLAR executor calculates a join order *resistance*. This quantity comprises the sum of intermediate results I observed in the current routing iteration, the number of tuples routed to the current join order T, and a constant w representing the overhead of a routing iteration without intermediate results. We calculate the resistance as $r = \frac{I}{T} + w$. The constant w prevents edge cases of a routing iteration with zero intermediates counting as infinitely better than an iteration with few intermediates. If only a few tuples are routed to a join order, the resistance may not be representative for a larger set of tuples. Consequently, the executor applies a moving average from previous iterations for smoothing fluctuations.

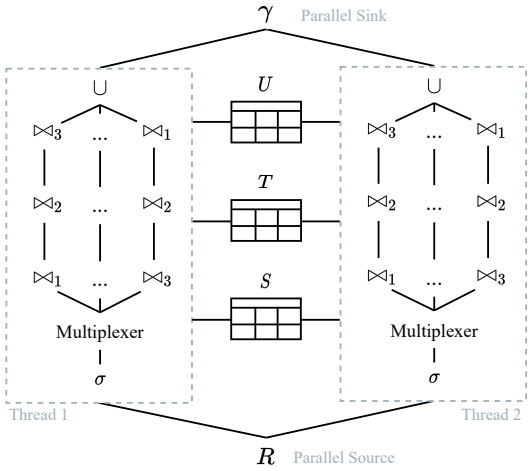

Figure 3: Parallel Execution: Each thread processes a pipeline with an isolated state, sharing build sides to probe into.

Parallel Execution: Similar to traditional data-parallel processing, POLAR executes pipelines in a multi-threaded fashion using multiple executors with thread-local states (cf. Figure 3). The executors concurrently fetch batches of tuples from the source and push results to the sink. Each executor has an isolated multiplexer state and calculates the path resistances solely based on its fetched tuples. Note that the executors only isolate their processing states but share build-side data structures such as read-only hash tables. With the input tuples, the executor follows the paths sequentially and independently from each other. The lack of global multiplexing may delay finding better paths because each multiplexer computes resistances individually. However, this parallelization strategy avoids synchronization among the executors and ensures correct resistance metrics for clustered data. As an alternative baseline parallelization strategy, POLAR also supports spawning one thread per join order and applying a backpressure mechanism to route tuples in a self-scheduling manner (cf. Section 3.4).

3 ROUTING STRATEGIES

At the core of POLAR pipeline execution is the multiplexer operator that makes routing decisions for exploration and exploitation to determine the number of input tuples for each join order. To this end, the pipeline executor passes an input chunk to the multiplexer, which uses a *routing strategy* to return the next join path index and a subset of the tuples to route. The routing strategies attempt to follow the *plans of least resistance*, which is the—potentially temporally changing—sequence of join order paths that incur the fewest number of intermediates. In this section, we first discuss the overall multiplexer algorithm, followed by four dedicated routing strategies used by the multiplexer and one self-scheduling strategy.

3.1 Overall Multiplexer Algorithm

Algorithm 3 shows the overall multiplexer algorithm. In the initialization phase, the multiplexer sends a small number of tuples to each join order with a resistance of zero (i. e., an uninitialized join order without reported resistance) to measure initial performance. When all join orders are initialized, the multiplexer requests a tuple distribution from a configurable routing strategy. The distribution indicates the fraction that each join order receives from the current input chunk. The algorithm finally returns the join order index with the largest fraction and its respective input tuple count. If the multiplexer does not emit all tuples from the input chunk, the multiplexer emits the remaining tuples from the tuple distribution in the next iteration until the input chunk is fully processed (cf. Section 2.4). In the following, we introduce four different routing strategies implementing the tuple distribution method. Assuming that POLAR is executed in a vectorized database system, the implementation must trade-off path optimality (following the cheapest path through the join orders) and operator cache friendliness (minimizing the number of join order switches). The caching aspect can impact the processing performance as the pipeline executor must flush all operator caches buffering intermediates for vectorization before reporting the join order resistance to ensure that each input tuple has been fully processed and was thus correctly counted. Additionally, processing without buffering or too frequent plan switches may increase the number of instruction cache misses [88].

Algorithm 3 Multiplexer

Input: Tuple Distribution T, Resistances $\mathbf{W}$
Output: Join Order Index *idx*, Tuple count c

1: **if** $\exists t_{idx} \in T : t_{idx} > 0$ **then**
2: $\quad$ *fraction* $\leftarrow t_{idx}$ $\quad$ *// Route tuples from previous multiplexing*
3: $\quad t_{idx} \leftarrow 0$
4: $\quad$ **return** *idx*, *fraction* $\cdot$ *INPUT_SIZE*
5: **else if** $\exists w_{idx} \in \mathbf{W} : w = 0$ **then** $\quad$ *// Initialize join order*
6: $\quad$ **return** *idx*, INIT_COUNT
7: $T \leftarrow$ GetTupleDistribution($\mathbf{W}$)
8: *idx*, *fraction* $\leftarrow$ max(T)
9: $t_{idx} \leftarrow 0$
10: **return** *idx*, *fraction* $\cdot$ INPUT_SIZE

3.2 Static Selection

Static routing strategies, or path selection approaches, do not perform any exploration apart from initialization. For this reason, they are very simple to implement, and thus, we omit their pseudo-code of GetTupleDistribution but provide high-level descriptions.

R1 InitOnce: This simple strategy retrieves the join order with the lowest resistance after the initialization phase and then routes every following chunk to that join order. InitOnce is extremely cache-friendly (i.e., in terms of tuple buffering in operator pipelines) because it does not require path switching or counting intermediates throughout the query. However, this strategy is prone to bad routing if a join order only performs well for the first few tuples. Moreover, it is unable to find well-performing paths if different join orders are optimal for different data clusters of the source.

R2 Opportunistic: The Opportunistic routing strategy is similar to InitOnce but makes use of the resistance reports after routing each chunk. If the reported resistance of the current join order exceeds the resistance of another, it routes the next input chunks to that join order. This approach allows switching join orders if the previous order deteriorates. However, the decision is solely based on the resistance of a single join order and old initialization results of others, which may miss additional opportunities, such as better plans for clusters of data. Cache flushing is needed but can be reduced by only updating the resistance after every n-th chunk, trading granularity with cache-efficiency.

3.3 Pro-active Exploration

In order to handle varying data characteristics and find well-performing join orders, routing strategies need to pro-actively explore alternative join orders. To this end, these strategies periodically route tuples to join orders that performed sub-optimal in the past. We introduce two strategies that both use the notion of an *exploration budget* that probabilistically bounds the overhead the strategy allocates for finding the optimal join path (cf. Section 2.1). The two following strategies use the exploration budget to calculate a tuple distribution over the join orders, producing additional intermediates based on the resistances measured so far.

R3 AdaptTupleCount: For each input chunk, the AdaptTupleCount strategy determines a tuple distribution that (assuming previous resistances) would stay within the exploration budget relative to the best path's intermediate count. The strategy initializes the output tuple fractions with ones (line 1), orders paths by

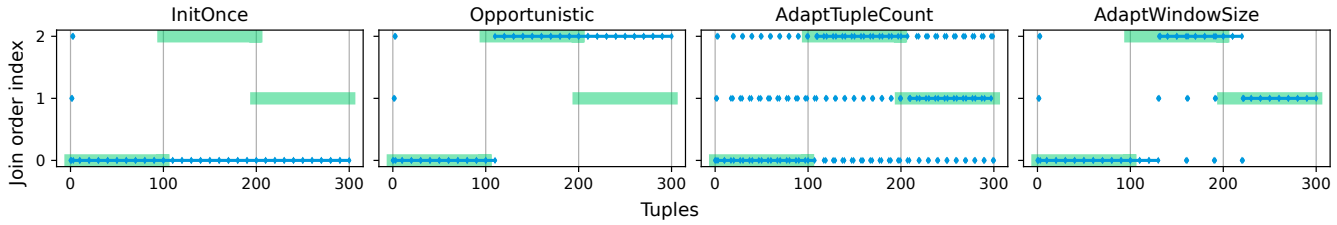

Figure 4: Behavior of four different routing strategies in a single-threaded example scenario with 30 input batches of 10 tuples each (green: optimal join paths, blue diamond: multiplexer invocations, blue line: path chosen by routing strategy).

Algorithm 4 GetTupleDistribution – AdaptTupleCount

Input: Resistances $\mathbf{W}$, **Output:** Tuple Distribution T

1: $T \leftarrow$ Initialize($|\mathbf{W}|$, 1)
2: $\mathbf{W} \leftarrow$ SortInc($\mathbf{W}$)
3: $cost \leftarrow$ LastElement($\mathbf{W}$)
4: **for** $i \leftarrow |\mathbf{W}| - 1$ **to** 0 **do** *// Calculate distribution bottom-up*
5: $\quad target \leftarrow w_i \cdot (1 + \text{BUDGET})$
6: $\quad decrease \leftarrow \frac{w_i - target}{w_i - cost}$
7: $\quad$ **for** $j \leftarrow i + 1$ **to** $|\mathbf{W}|$ **do** *// Adjust fractions to new target*
8: $\quad\quad t_j \leftarrow t_j \cdot decrease$
9: $\quad t_i \leftarrow 1 - decrease$
10: $\quad cost \leftarrow target$
11: **return** T

Algorithm 5 GetTupleDistribution – AdaptWindowSize

Input: Resistances $\mathbf{W}$, **Output:** Tuple Distribution T

1: $T \leftarrow$ Initialize($|\mathbf{W}|$, 0)
2: $t_{\text{MININDEX}(W)} \leftarrow 1$ *// Route all tuples to least resistant order*
3: *size*, *offset* $\leftarrow$ GetWindowState()
4: **if** *size* = 0 **then** *// Determine window size*
5: $\quad W' \leftarrow \mathbf{W} \setminus \text{MIN}(\mathbf{W})$
6: $\quad size \leftarrow \frac{1}{\text{INPUT\_SIZE}} \cdot \frac{\sum w'_i \cdot \text{INIT\_COUNT}}{\text{BUDGET} \cdot \text{MIN}(W)}$
7: **if** *offset* < *size* **then**
8: $\quad offset \leftarrow offset + 1$
9: **else** *// Re-explore join orders next time*
10: $\quad \mathbf{W} \leftarrow$ SetToZero($\mathbf{W} \setminus$ MIN($\mathbf{W}$)) *// Reset resistances*
11: $\quad offset, size \leftarrow 0$
12: SetWindowState(*size*, *offset*)
13: **return** T

their resistances (line 2), and adjusts the distribution in a bottom-up fashion. We start by reducing the problem to the two join orders with the highest resistances. The target cost is based on the lower resistance and the exploration budget, which we then use to calculate a *decrease* factor (line 6). By multiplying that factor with the join order's tuple fraction, we find the amount of tuples that must be sent to the worst join order to stay within the exploration budget based on the resistance of the second-last join order. Consequently, the amount for the second-last order is 1 − *decrease* (line 10). We calculate these values for the next-best join order while decreasing the fractions of the orders with higher resistances until we reach the first join order. AdaptTupleCount calculates tuple distributions for each incoming chunk, allowing for fine-grained path exploration bounded by the exploration budget. However, splitting each chunk into smaller sets of tuples obstructs vectorized execution and causes many cache flushes to report resistances. To reduce the number of splits per chunk, the algorithm could serve only join orders receiving more than n tuples, redistributes unserved tuples, and recalls them when calculating the next distribution.

R4 AdaptWindowSize: In contrast to AdaptTupleCount, the AdaptWindowSize strategy does not calculate individual tuple counts per join order. Instead, this strategy either routes complete chunks or a static, low number of tuples and adapts the window size (i. e., number of input chunks) within which it will only serve the best join order. When exceeding the window, it re-initializes the remaining join orders by routing a small number of tuples to each of them. The internals of this strategy are shown in Algorithm 5, which distinguishes between three cases. Irrespective of the case, this strategy always returns a tuple distribution in which the best order receives the whole input chunk (line 2). If there is no window size calculated yet (line 4), we estimate the number of intermediates occurring in a reinitialization phase based on the resistances and adjust the number of tuples for reinitialization accordingly. We then divide the cost estimate by the overhead allowed by the exploration budget resulting in the number of tuples that should be routed to the best join order before reinitializing the others. The number of tuples per chunk scales this value down to the window size (line 6). If the current offset does not exceed the window size, we increment it (line 9). Finally, if the window size is exceeded, we set the resistances for all join orders, except the best, to zero (line 12) to trigger re-initialization on the next invocation. By routing multiple chunks to the same join order, the AdaptWindowSize strategy better allows for vectorization and can defer cache flushing until the window size is exceeded. On the other hand, its path exploration granularity is more coarse-grained than AdaptTupleCount as changes in resistances may appear within the routing window.

Example 1 (Routing Example). *To illustrate the behavior of the routing strategies, Figure 4 compares their decisions in an example scenario (three join orders, 30 input chunks of 10 tuples each). The resistances of the join orders change every 10 chunks, namely* $C_0 \leftarrow \langle 1, 10, 15 \rangle$, $C_{10} \leftarrow \langle 10, 15, 5 \rangle$, *and* $C_{20} \leftarrow \langle 10, 1, 5 \rangle$. *The resulting optimal path changes from 1st to 3rd to 2nd (indicated as green solid lines). The blue diamonds show the multiplexer invocations of the different paths, and the blue line indicates the chosen path.* InitOnce *tests every join order once and follows the initially optimal path. The* Opportunistic *strategy switches to the correct path after the first one deteriorates but misses the second switch.* AdaptTupleCount *correctly adapts to the optimal join paths but requires many switches and multiplexer invocations, including cache flushing. Finally,* AdaptWindowSize *first uses a large window and thus detects*

the join order switch only after a delay. Later, it reduces the window size as the difference between the resistances decreases. Hence, the next switch comes after a shorter delay due to window resizing.

3.4 Self-scheduling

R5 Backpressure: To enable comparing against a self-scheduling strategy without parameters, we include a "backpressure" strategy. Instead of using a multiplexer, the POLAR pipeline spawns individual threads for each join order so that each thread concurrently pulls new input chunks. The approach does not depend on a budget and simply favors faster join orders as they can pull more chunks per time unit than others. The Backpressure strategy does not rely on multiplexing, does not require any operator cache flushes, and thus, fully supports vectorized execution. However, with potentially many join orders, this strategy has the intrinsic limitation that the majority of threads waste CPU cycles on sub-optimal paths.

4 LIMITATIONS AND SCOPE

POLAR is designed for adaptive query processing with non-invasive system integration as well as small and bounded overhead. This design leads to certain limitations as its applicability is ultimately dependent on the system's existing query optimizer.

- *Amenable Pipelines:* POLAR only replaces left-deep-trees. If the optimizer emits a right-deep tree (where intermediates feed into the build side of hash joins), POLAR cannot generate alternative join orders, as there are no pipelines with more than one join. Support for directed acyclic graphs (DAGs) and bushy trees is interesting future work.
- *Source Table:* As POLAR replaces normal operator pipelines that consume tuples from a specific source, it cannot change the source (sometimes called driver [66]) table.
- *Operator Types:* POLAR only supports pipelines with join sequences. Extended support for additional operators—such as projection, selection, and groupjoin [75]—is interesting future work as well.

POLAR is currently most applicable to star-schema workloads with cardinality estimation challenges (e. g., parameterized queries, UDFs, correlated data). If the source table contains clustered data (e. g., orders sorted by date with different join cardinalities over time), POLAR can further exploit different plans for different data.

5 EXPERIMENTS

Our experimental evaluation of a prototype implementation of POLAR studies its general applicability and performance characteristics with a variety of benchmarks. After describing the prototype and experimental setting, we use various micro benchmarks to evaluate the behavior of different strategies and parameters and conduct end-to-end performance comparisons with DuckDB [81], Postgres [90], SkinnerDB [93, 94], SkinnerMT [97], and Lookahead Information Passing [102] in DuckDB. Our major findings are that:

(1) Non-invasive AQP yields robust end-to-end performance,
(2) Offers substantial speedups for certain queries, especially on skewed benchmarks, and
(3) Is already effective with small exploration budgets of ≤ 1 %.

Table 2: Total Number of Intermediates and Fraction of Total Execution Time Spent in Amenable Pipelines (Coverage).

Benchmark	DuckDB	Routing	Static	Coverage
JOB	107.49 M	16.92 M	25.84 M	36 %
SSB	747.67 M	547.40 M	578.23 M	73 %
SSB-skew	2690.05 M	342.67 M	474.28 M	99 %

5.1 Experimental Setup

Prototype: We implemented POLAR in DuckDB, a state-of-the-art OLAP DBMS. The prototype[1], including the different routing and selection strategies, is 2500 LoC and requires minimal changes to the existing DuckDB code. The input to POLAR is a query plan produced by the DuckDB optimizer, which uses equivalence sets to estimate cardinalities [36] and DPhyp [74] for join ordering.

Evaluation System: We conducted all experiments on a Lenovo ThinkSystem SR635 server with a single AMD EPYC 7443P CPU at 2.85 GHz (24 physical/48 virtual cores), 256 GB DDR4 RAM at 3.2 GHz, 1×480 GB SATA SSD, 8×2 TB SATA HDDs (data) and Mellanox ConnectX-6 HDR/200 Gb Infiniband. We compiled the source code with clang-12 on Ubuntu 20.04.1.

Benchmarks: We evaluate POLAR on the Join Order Benchmark (JOB) [64], Star Schema Benchmark (SSB) [79], and a modified SSB with correlation and skew (SSB-skew), both with a scale factor of 100. We introduce SSB-skew as a benchmark with SSB's applicability (long join pipelines, cf. Section 5.2) but real-world data characteristics such as cross-correlations, skew, and clustered data. In SSB-skew, most customers are from the US and most suppliers are from Asia. Moreover, the `part` table references are skewed, so many lineitems belong to a few part categories and brands. The `lineorder` table is clustered by order date, with a recession of few orders in 1997 and a year of growth in 1998. Finally, the number of lineitems from suppliers in the United States increases over time. SSB-skew uses SSB's original queries except query set 1 (only one join). We do not use TPC-H [92] and JCC-H [20] because (1) their optimal join orders are well-known and often tuned for, and (2) the join orders generated by DuckDB are mostly right-deep trees, not amenable to POLAR. We also considered the LDBC Social Network BI Workload [7], but LDBC requires advanced SQL features which are not supported by all systems we compare, and the DuckDB optimizer often generates pipelines with alternating joins and projections, which our prototype does not yet support (cf. Section 4).

5.2 Applicability Study

In a first series of experiments, we aim to analyze the potential of POLAR by estimating the possible reduction in the number of intermediates with optimal routing strategies. These results serve as a baseline—in terms of an upper bound—for later experiments evaluating the quality of POLAR routing strategies. Furthermore, we also examine the time spent in amenable pipelines. Combining these measures allows estimating the overall potential.

Potential Reduction of Intermediates: Table 2 shows the potential performance improvements achievable with adaptive join order switching. We calculated the potential improvement by measuring the number of intermediate results for all amenable pipelines

[1]We share our prototype and artifacts at https://github.com/damslab/reproducibility.

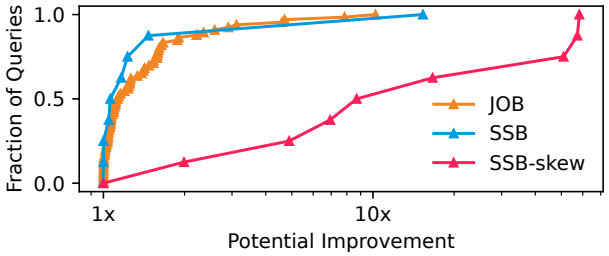

Figure 5: Potential Query Performance Improvement.

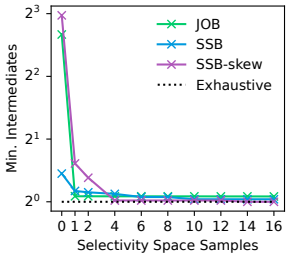

Figure 6: SelSampling – Relative Number of Intermediate of Best Join Order with Increasing Number of Samples.

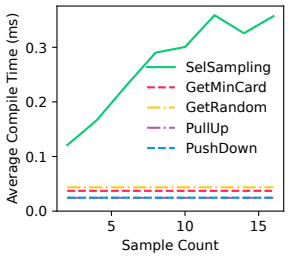

Figure 7: Average Pipeline Compilation Time [ms] for Different Join Order Selection Strategies on SSB.

using DuckDB's default join order, the best static join order for each pipeline, and an estimated optimal routing strategy. We determined this optimal value using a multiplexer debugging mode, routing each input chunk to every legal join order and measuring the minimal number of intermediates. Table 2 shows that dynamic tuple routing reduces the number of intermediates for all benchmarks but the potential improvement over an ideal static join order is most significant with skewed datasets. For JOB, the best static join order produces 25.84 M tuples compared to 107.49 M tuples from DuckDB's default join order. Dynamic routing further improves this number to 16.92 M tuples. This result shows that a better static join order could improve JOB to a large extent without dynamic join order switching at runtime. For SSB, neither join order switching nor better join orders considerably improve the number of intermediates. The best static join order produces 578.23 M tuples, a moderate improvement over the 747.67 M tuples in the default DuckDB plan, and dynamic routing yields only a slight additional improvement to 547.40 M tuples. DuckDB's near-optimal plan for SSB is expected because the benchmark contains well-behaved FK/PK joins and uniform, non-skewed data. For SSB-skew, which includes correlation and skew, the potential for improvement is much higher. The default DuckDB join order produces 2690.05 M tuples, while the best static join order produces 474.28 M tuples. Tuple routing further decreases the optimal number of intermediates to 342.67 M. Thus, SSB-skew benefits from routing about the same as JOB.

Coverage of Amenable Pipelines: Given DuckDB's default query plans, we measure each benchmark's total execution time and the time spent processing POLAR-amenable pipelines (i. e., pipelines containing left-deep trees of two or more joins). Comparing the difference of these values yields the *Coverage*, that is, the fraction of time spent in improvable pipelines. Note that the coverage also includes other operators from these pipelines, such as scans and aggregations, which POLAR cannot improve. The Coverage column in Table 2 shows that for JOB, DuckDB only spends 36 % of the processing time in POLAR-applicable pipelines. Consequently, almost two-thirds of the total execution time cannot be improved by POLAR. For SSB, DuckDB spends 73 % of the time in applicable pipelines, but many of them are dominated by large table scans and the joins only account for a small portion of the time. For SSB-skew, DuckDB spends almost all of the execution time (99 %) in applicable pipelines, and joins account for a large fraction of that time, providing substantial room for performance improvements.

Potential Query Performance Improvement: Furthermore, we aim to assess how much the runtime of individual queries could be improved. We estimate these improvements per query by multiplying the optimal number of intermediates with the coverage of amenable pipelines (assuming a linear relationship between tuple count reduction and execution time). Figure 5 shows a cumulative distribution function over the potential performance improvements for all queries. Less than 10 % of the queries in JOB and SSB can be improved by more than 2x. Queries in SSB-skew show a much larger potential for improvement: over 40 % of the queries can be improved by more than an order of magnitude.

5.3 Micro Benchmarks

In order to understand the trade-offs of different join selection and routing strategies, in a second series of experiments, we conduct several micro benchmarks regarding plan quality, compilation time, adaptivity, and runtime overhead. We also investigate the impact of the exploration budget on adaptivity and overhead. All micro benchmarks were executed single-threaded to isolate the effects.

Sampling for Join Order Selection: The SelSampling join selection strategy—introduced in Section 2.3—systematically samples the space of cardinalities to find alternative join orders. We set the maximum join order count *MAX* to $|J|!$ (i. e., the number of possible orders for joins J, assuming clique queries). For an increasing number of samples, we determine the best possible outcome for the resulting join order set in terms of the intermediate optimum introduced in Section 5.2. As a baseline, we exhaustively enumerate all possible join orders for each of the join pipelines. The results in Figure 6 show that even few samples allow for effective intermediate result reduction close to the level of exhaustive enumeration. By default, we set the sample count to 8 for the following experiments, which allows POLAR to find well-performing sequences of join paths while excluding unnecessary routing options. We used SSB-skew to test performance penalties induced by exhaustive enumeration and found deteriorations—despite no very long join pipelines—of up to 9 % with AdaptTupleCount and 1.5 % with AdaptWindowSize compared to SelSampling using 8 samples.

Pipeline Compilation Time: The time required to compile a POLAR pipeline is dominated by the join order selection. We measure this compilation time for the different join order selection strategies using pipelines with varying numbers of joins from SSB and SSB-skew as they contain the longest join pipelines. Figure 7 compares the average pipeline compilation time with an increasing number of samples for SelSampling to the other strategies (again, with a join path limit of 8). Our results show that SelSampling generally takes longer to compile than the baselines, and its compilation time increases linearly until it reaches 8 join orders. After

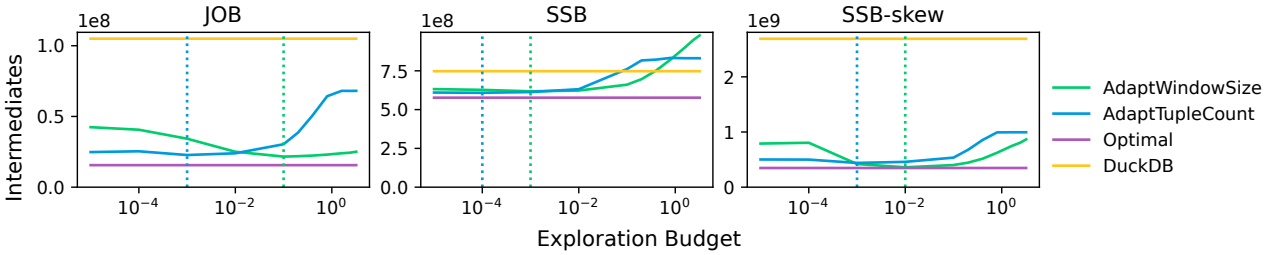

Figure 8: Exploration Budgets – Number of Intermediate Tuples for Different Routing Strategies and Exploration Budgets (the dotted lines denote sweet spots in which the strategy generates minimal intermediates).

Table 3: Join Order Selection – Total Number of Intermediates for POLAR Pipelines with Different Selection Strategies.

Enumeration	JOB	SSB	SSB-skew	JOB-ldt
DuckDB	107.49 M	747.67 M	2690.05 M	248.30 M
Optimal	16.92 M	547.40 M	342.67 M	—
SelSampling	17.97 M	576.76 M	347.61 M	160.52 M
GetMinCard	16.92 M	573.72 M	351.42 M	192.68 M
GetRandom	16.92 M	572.31 M	356.09 M	193.07 M
PushDown	17.04 M	562.50 M	356.09 M	208.41 M
PullUp	17.22 M	595.18 M	512.69 M	210.57 M

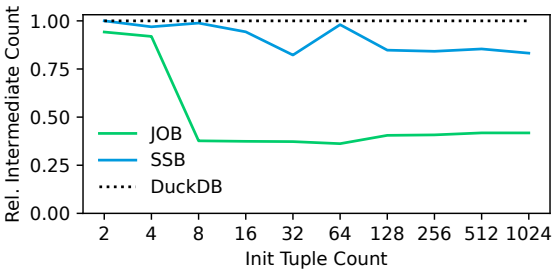

Figure 9: Path Initialization – Relative Number of Intermediates for Initial Input Tuple Counts using InitOnce.

that, the increase becomes smaller as the strategy stops taking new samples if we already found *MAX* distinct join orders. In any case, the compilation time for any pipeline is way below one millisecond and thus, negligible compared to the overall query processing times.

Join Order Selection: We further compare the quality of the join order selection strategies by comparing the actual number of intermediates to the optimal number, as described in Section 5.2. To stress-test the different join order selection strategies, we also compared *JOB-ldt*, which compiles the JOB queries using a greedy algorithm that only generates left-deep trees. Table 3 summarizes the results. For GetMinCard and GetRandom, we limit the number of join orders to 8, analogous to the SelSampling. For our set of benchmarks, even simple strategies such as PushDown yield competitive results that are close to the optimal number of intermediates. However, especially on SSB-skew, PullUp performs poorly. On JOB-ldt, SelSampling outperforms our baselines noticeably. Furthermore, as GetMinCard and GetRandom always produce the maximal number of join orders within the user-set limit and PushDown may include plans that are highly unlikely well-performing (such as an PK-FK join on a table without predicate as first join), SelSampling is the only strategy that may select small sets of useful, complementary join orders excluding unreasonable join paths. Therefore, we use SelSampling as POLAR's default.

Join Order Initialization: All routing strategies (excluding Backpressure) use a static number of tuples to initialize each of the join paths. In AdaptWindowSize, this number is also used to explore weaker paths after the initialization phase. To find a meaningful tuple count for path (re-)initialization, we conduct an experiment using the InitOnce strategy with an increasing number of initialization tuples and count the resulting number of intermediates. Figure 9 shows the results of that experiment. For JOB, 8 initialization tuples are already sufficient to determine join orders that produce less than 40 % intermediates incurred by DuckDB. However, for SSB, the number of intermediates gradually improves with the number of tuples. Therefore, we conservatively set the default initialization tuple count to 1024, which is also the chunk size in our DuckDB-based prototype.

Exploration Budgets: For adaptive routing strategies, such as AdaptTupleCount and AdaptWindowSize, the quality of routing depends on the exploration budget. A higher budget allows the strategies to adapt better to path changes but also incurs larger overheads. To understand how the exploration budget affects the different workload characteristics, we execute JOB, SSB, and SSB-skew with exploration budgets from 0.001 % to 320 %. Figure 8 shows how the number of intermediates produced by the two adaptive routing strategies varies with increasing exploration budget. Both AdaptTupleCount and AdaptWindowSize achieve close to the optimal numbers of intermediates. However, the ideal exploration budgets differ for each benchmark. We attribute this effect to the differences in exploration potential, already observed in 5.2. While a larger budget only creates exploration overhead for SSB, exploration is crucial to find better join order alternatives for JOB and SSB-skew as the data characteristics change during query execution. AdaptWindowSize is especially sensitive to the exploration budget while AdaptTupleCount shows more robust behavior: small exploration budgets of up to 1 % are sufficient to obtain robust performance characteristics. Moreover, the sweet spots for AdaptWindowSize are generally higher than for AdaptTupleCount as the latter consistently keeps exploring alternative join orders even under small exploration budgets. As a result of this analysis, we set the exploration budget for AdaptWindowSize to 1 % and AdaptTupleCount to 0.1 % for the following experiments.

Routing Strategies – Intermediates: We compare the different routing strategies from Section 3 with regard to the number

Table 4: Intermediate Results – Number of Intermediates of Different Routing Strategies (Tuned Exploration Budgets).

Routing Strategy	JOB	SSB	SSB-skew
DuckDB	107.49 M	747.67 M	2690.05 M
Optimal	17.97 M	576.76 M	347.61 M
InitOnce	44.92 M	622.28 M	790.52 M
Opportunistic	24.81 M	**608.03 M**	499.75 M
AdaptTupleCount	**22.72 M**	613.07 M	439.65 M
AdaptWindowSize	25.04 M	622.74 M	**360.12 M**
Backpressure	88.21 M	2542.49 M	3940.55 M

Table 5: Execution Time – Pipeline Execution Time of Different Routing Strategies [seconds].

Routing Strategy	JOB	SSB	SSB-skew
DuckDB	48.85	92.88	147.32
InitOnce	31.43	93.13	86.26
Opportunistic	31.16	107.28	92.25
AdaptTupleCount	31.43	120.51	100.85
AdaptWindowSize	**30.85**	**94.19**	**76.28**
Backpressure	67.69	217.93	222.76

of intermediates they produce. Table 4 shows the results. AdaptTupleCount produces the least intermediates for JOB and shows competitive performance for SSB, whereas AdaptWindowSize produces the fewest intermediate results for SSB-Skew. InitOnce performs substantially worse than the adaptive strategies because it picks sub-optimal join orders whenever the initialization phase is not representative for the remaining data batches. This observation is especially pronounced for SSB-skew, where there are different optimal plans for different clusters of the data. The Opportunistic strategy performs best on SSB as it allows path switching without the overhead of exploration. Backpressure produces the most intermediate results because many executor threads process tuples through sub-optimal join orders.

Routing Strategies – Execution Time: The performance of routing strategies does not solely depend on reducing intermediates. Another influential factor is how much adaptivity negatively affects vectorized execution. A high path switching rate can reduce intermediate buffer sizes, as explained in Section 3. Therefore, we also examine the actual pipeline execution times for each routing strategy, as they are closely correlated with overall query execution time. Table 5 shows the total pipeline execution time for different strategies using the tuned exploration budgets reported before. Since AdaptWindowSize trades path exploration granularity for better vectorization, the strategy performs best for exploration budgets that are below its sweet spots for minimal intermediates. Interestingly, the lowest number of intermediates does not necessarily lead to the lowest pipeline execution time. Despite producing more intermediates than its competitors on JOB and SSB, AdaptWindowSize outperforms each of them in all of our benchmarks. Therefore, we conclude that the strategy offers the best trade-off between reducing the number of intermediates and preserving good vectorized execution. Thus, we use AdaptWindowSize as POLAR's default routing strategy for the remainder of the experiments.

Table 6: Overall Performance Impact – Single-threaded Total Execution Time, and Max Query Execution Time [seconds].

	Total Execution Time			Max. Query Time		
	JOB	SSB	SSB-skew	JOB	SSB	SSB-skew
DuckDB	134.2	127.9	148.8	10.8	17.0	35.6
POLAR	**115.4**	**129.0**	**76.8**	**3.9**	**18.3**	**11.9**
Speedup	**1.16x**	0.99x	**1.94x**	**2.78x**	0.93x	**2.98x**

5.4 End-to-End Performance Comparison

Informed by the results from our micro benchmarks, we evaluate POLAR's end-to-end benchmark performance with SelSampling join order selection with 8 samples, an initialization tuple count of 1024, and the AdaptWindowSize routing strategy with a 1 % exploration budget for all benchmarks. In this context, we compare POLAR with DuckDB [81], a Lookahead Information Passing [102] prototype on DuckDB, Postgres [90], SkinnerDB (i. e., Skinner-C) [93], and SkinnerMT [97], a state-of-the-art AQP system, in both single- and multi-threaded configurations.

Overall Performance Impact: Table 6 shows the total, end-to-end, single-threaded execution time and the maximum query execution times for DuckDB and POLAR on the different benchmarks. POLAR shows a slight overhead on SSB in total and maximum execution times and a moderate total execution time improvement of 1.16x on JOB. Moreover, POLAR shows a substantial 1.94x end-to-end improvement for SSB-skew and reduces the maximal query runtimes for JOB and SSB-skew by 2.78x and 2.98x, respectively. These results show that POLAR yields robust performance with substantial improvements for workloads on skewed data and only minor overhead for workloads where adaptation is not needed.

Query Performance Impact: Figure 10 shows the speedups and slowdowns for each individual query with POLAR-amenable pipelines in JOB, SSB, and SSB-skew. A value of 1 indicates an improvement of 100 % (i. e., half the execution time or 2x), whereas a value of -1 indicates double the execution time. For most JOB queries, POLAR has no positive or negative effect on the execution time. However, for a few queries, POLAR substantially reduces the execution time by up to 9x. The two queries with the largest speedups are also the longest-running queries in the benchmark. On SSB, the POLAR overhead increases the execution time for most queries, as expected, given how close to optimal the original plans are. However, one of the SSB queries also benefits from POLAR. Finally, almost all queries in SSB-skew improve with POLAR, up to 4x in one case. Therefore, POLAR achieves substantial performance and robustness improvements on non-uniform data and incurs only slight overhead for some queries on uniform data due to plan exploration and impact on vectorization at runtime. However, this moderate overhead is an acceptable price to pay for increased robustness, and the overhead could be further decreased when specializing POLAR to the underlying runtime system.

Number of Intermediates: The execution times of our baseline systems are strongly correlated with their underlying execution engines. Hence, we first conduct an experiment comparing the number of intermediates produced by their join orders. Table 7 shows the results. Note that SkinnerDB ran out-of-memory for SSB and SSB-skew. However, for JOB, SkinnerDB achieves a much lower number

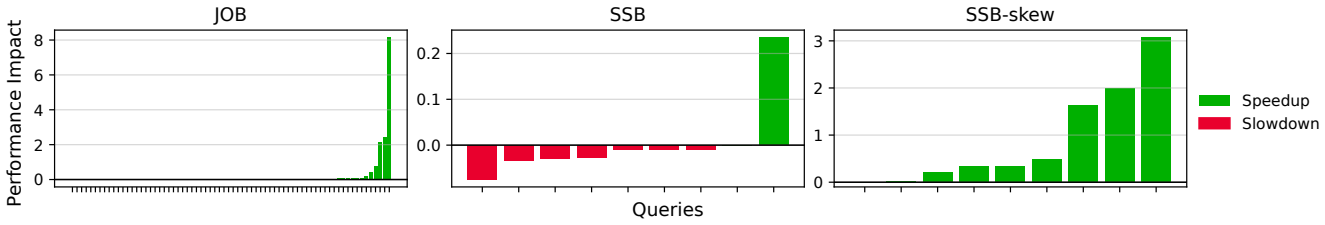

Figure 10: Individual Query Performance Impact – Query performance changes between unmodified DuckDB and POLAR. A value of +1 indicates the query was 100 % faster (2x), and a value of -1 indicates a 100 % overhead (doubled execution time).

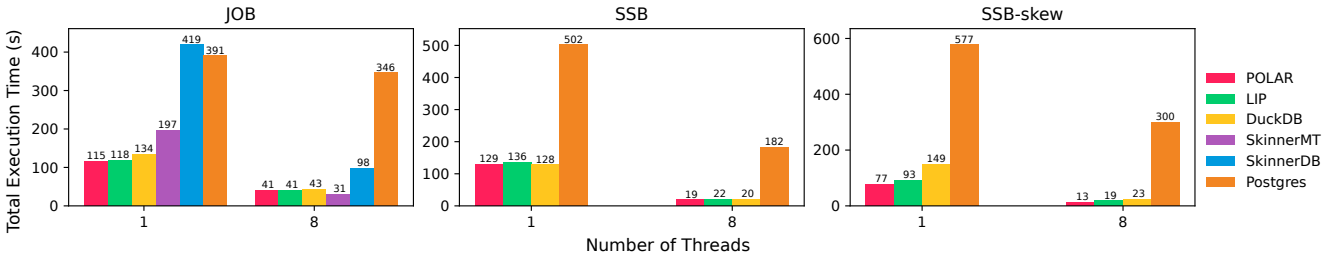

Figure 11: AQP System Comparison – Total execution times for JOB, SSB, and SSB-skew, using 1 and 8 threads [seconds].

Table 7: Total Number of Intermediates (C_{out}) Comparison.

System	JOB	SSB	SSB-skew
DuckDB	896 M	761 M	2690 M
Postgres	272 M	598 M	503 M
SkinnerDB	121 M	—	—
POLAR	685 M	636 M	360 M

of intermediates than DuckDB, POLAR, and Postgres because it is not restricted to amenable pipelines. The comparison further shows that Postgres generally produces plans with fewer intermediates than DuckDB. Using POLAR reduces DuckDB's intermediates and outperforms Postgres by 1.4x on SSB-skew.

AQP System Comparison: To contextualize POLAR's performance, we compare POLAR, DuckDB, Postgres, SkinnerDB, and SkinnerMT, measuring the total execution time in single- and multi-threaded (8 threads) configurations. Additionally, we implemented a prototype of Lookahead Information Passing [102] (LIP) in DuckDB to compare POLAR against a bitmap filtering approach. We allow SkinnerDB/MT to cache indexes on all join columns in memory to reduce its pre-processing time. We use Postgres 12.15 with indexes on all foreign key columns to prevent it from compiling aggressive plans with fatal nested-loop joins on unindexed join columns. Figure 11 shows that POLAR performs equally or better than DuckDB, SkinnerDB, and Postgres on almost all benchmarks. On SSB, POLAR shows a similar performance as DuckDB both single- and multi-threaded. On SSB-skew, POLAR outperforms DuckDB by 1.9x single-threaded and by 1.8x multi-threaded. While LIP shows equal execution times as POLAR on JOB, it performs slightly worse on SSB and substantially worse on SSB-skew. Regarding individual JOB queries, LIP shows occasional performance regressions of more than 2x, which is in line with experimental results from other bitmap filtering approaches [31, 63]. We account this performance gap to the large number of additional hash probes, lack of vectorization, and unstable bloom filter orderings. SkinnerMT shows the best performance on JOB with multiple threads. However, SkinnerDB/MT run out-of-memory on SSB and SSB-skew. Previous experiments on SSB with scale factor 10 showed POLAR speedups of up to 15.6x over SkinnerDB (8 threads, 1.9 s vs 29.6 s) and around 3.9x over SkinnerMT. We attribute this to SkinnerDB/MTs custom join operator with tuple-at-a-time processing and the systems' ability to change the source table, which requires building hash tables for each relation. As expected, Postgres shows much higher total execution times than POLAR and DuckDB, given its different target workloads and runtime system. These experiments demonstrate the benefits of a non-invasive, bounded-overhead system design in an engine designed for analytic workloads. In contrast to invasive AQP systems, POLAR favors reusing existing database components and original plans resulting in competitive performance and much greater performance robustness with modest overhead.

6 RELATED WORK

Besides the brief background in the introduction, here, we broadly survey related work and emphasize the differences of POLAR.

Traditional AQP: Adaptive query processing has received lots of attention in the data management literature, and great surveys [10, 30] and tutorials [29, 53] already exist. Existing classifications distinguish AQP for traditional ad-hoc queries (or plan-based systems) versus continuous queries [10], as well as a spectrum of adaptivity from coarse- to fine-grained adaptation [29]. First, *inter-query* optimization utilizes learned cardinalities for expressions [23, 25, 89] for optimizing future queries. Second, *inter-pipeline* optimization utilizes monitored statistics even within a single query. Examples are late binding (staged execution) with re-optimization at pipeline breakers [29] or at parameter binding in parametric query optimization [17]. Third, *inter-operator* re-optimization compiles new remaining plans at so-called checkpoint operators [57]. Similar,

progressive and pro-active re-optimization apply plan changes if actual cardinalities are outside computed validity ranges [11, 71]. Fourth, *intra-operator* adaptivity allows changing plans after partial operator evaluation. Examples are union stitch-up plans and handling of state in double-pipelined hash joins [54], intra-query adaptivity via reinforcement learning in SkinnerDB [93, 94, 97], as well as adaptive join reordering of index-nested-loop joins with depleted states for correctness [66]. Fifth, there is tuple routing with routing policies in Eddies [8, 9, 16, 28]. Many of these strategies require both optimizer and runtime extensions for effective and efficient adaptivity. Moreover, Eddies requires complex tuple tracking to produce correct results. Instead, POLAR aims at a simple system integration with bounded exploration overhead.

AQP for Continuous Queries: The adaptation of continuous queries—on infinite data streams—always focuses on *intra-operator* and *tuple-routing*. Early stream processing systems with adaptive query processing include STREAM [13], Aurora [2]/Borealis [1], NiagaraCQ [26], and TelegraphCQ [24]. Unique characteristics are the incremental collection of statistics to detect workload changes [12], multi-query optimization with queries entering and leaving the system, asynchronous optimization outside the critical path [18], the applicability of load shedding [91], as well as stateful operators and queues which require draining for plan changes [96]. Modern distributed stream processing engines like Flink [6], Spark [99], Beam [5], Heron [62]/Dhalion [37], and NebulaStream [100] further deal with the reconfiguration of distributed query topologies. In contrast to POLAR, AQP is naturally deeply integrated in almost all components of such stream processing engines.

Different Plans for Different Data: Both, ad-hoc and continuous queries are typically compiled and optimized according to average statistics. However, especially in correlated data, relations are naturally divided into partitions with different characteristics [95]. Early work on selectivity-based partitioning [80] and content-based routing [16] address this observation by generating different plans for different partitions, and different value-based routing policies. Such approaches often leverage more fine-grained statistics such as serial histograms (i.e., detailed frequency matrices) [51]. Recent work on multi-way join size estimation [55, 78] utilize hash-based translation grids [78] and AKMV sketches [15] as well as fast-AGMS sketches [55]. Since selectivity-based partitioning might compute the same intermediate multiple times, further work on sharing-aware horizontal partitioning [95] introduced a conditional join plan, and related optimization and runtime techniques. Exploratory AQP via reinforcement learning like SkinnerDB [93] would also lend itself to discovering different paths. Due to repeated path sampling, POLAR can also exploit different paths for different data, but only if these tuples are scanned in a clustered manner.

Micro Adaptivity: Besides finding alternative plans (e.g., join orders), some work also focused on micro adaptivity. Raducanu et al. introduced this concept in Vectorwise [82] by providing alternative kernel implementations (e.g., branch, no-branch), measuring their runtime on sample vectors, and selecting the best configuration via a learning algorithm. Later work used performance counters to minimize the measurement overhead, and more properties such as sortedness and co-clustering [101]. Other forms of micro adaptivity are compiling continuous queries for HW specialization, parallelization, as well as exploitation of selectivities or value distributions [39, 85]. Micro adaptivity requires specific optimization algorithms, whereas POLAR relies on existing optimizers without changes.

AQP for Non-relational Workloads: AQP ideas have also been applied and extended for non-relational systems and workloads. Examples include JIT compilation for programming languages [47] (e.g., Java or WebAssembly); lazy expression compilation in TensorFlow [77], dynamic recompilation of blocks and functions in SystemML [19]; periodic or on-demand reoptimization of integration flows [18]; and AQP on raw data [59]. These systems also incrementally collect telemetry and perform plan adaptation, but unlike POLAR, seek a single optimal plan or configuration.

Learned Optimizers and Estimators: With the trend towards ML for systems [61], there has been substantial progress on improved cardinality estimates that could mitigate some of the need for AQP. Early work focused on the featurization of schemas and workloads and sampling-based training data collection [35, 60, 98]. Hilprecht and Binnig later introduced representations for zero-shot cost models [45, 46] that can generalize to unseen database instances. Early work like LEO [89] also focused on learned cardinalities, but in retrospective had the problem of "fleeing from knowledge to ignorance" [67] because the exponential search space gets only sparsely sampled, and skewed cardinalities are often larger than the estimates under independence assumption. However, recent work has shown that learned optimizers and cardinality estimators can learn from mistakes [69, 70], making a case for stateful, learning-based systems [68], especially for cloud DBMS like Snowflake [27] or Redshift [40]. In contrast to POLAR, integrating learned optimizers and estimators is still very invasive in terms of system complexity, bootstrapping, and integration points.

7 CONCLUSIONS

We introduced the new concept of plans of least resistance for leveraging adaptive query processing in a non-invasive manner. POLAR pipelines replace, where applicable, standard join pipelines and internally multiplex tuple batches among alternative join paths. This design allows periodically sampling join paths, collecting telemetry, and adapting the routing to the best path accordingly. We draw three key conclusions. First, the simple design without optimizer changes greatly simplified the integration into systems such as DuckDB. Second, POLAR shows robust performance but only on workloads yielding a large fraction of applicable pipelines. Third, there are examples of substantial performance improvements for individual pipelines, queries, and workloads, especially for skewed data (fix for bad cardinality estimates) and clustered data (exploit different plans for different data partitions). Interesting directions of future work include more advanced strategies for selecting alternative pipelines (e.g., considering the uncertainty of cardinality estimates), and broader support for different plan structures (e.g., DAGs, bushy plans, additional operators like groupjoin [75]).

ACKNOWLEDGMENTS

We thank the participants of Dagstuhl Seminar 17222 [21] and 22111 [22] for inspiring this research and invaluable discussions. We also gratefully acknowledge funding from the German Federal Ministry of Education and Research (under research grant BIFOLD23B) as well as initial funding from SAP.

REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR*. 277–289. http://cidrdb.org/cidr2005/papers/P23.pdf

[2] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB J.* 12, 2 (2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z

[3] M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, and Jayant R. Haritsa. 2010. On the Stability of Plan Costs and the Costs of Plan Stability. *PVLDB* 3, 1 (2010), 1137–1148. https://doi.org/10.14778/1920841.1920983

[4] Ashraf Aboulnaga, Peter J. Haas, Sam Lightstone, Guy M. Lohman, Volker Markl, Ivan Popivanov, and Vijayshankar Raman. 2004. Automated Statistics Collection in DB2 UDB. In *VLDB*. https://doi.org/10.1016/B978-012088469-8.50100-5

[5] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[6] Alexander Alexandrov et al. 2014. The Stratosphere platform for big data analytics. *VLDB J.* 23, 6 (2014), 939–964. https://doi.org/10.1007/s00778-014-0357-y

[7] Renzo Angles et al. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299

[8] Remzi H. Arpaci-Dusseau. 2003. Run-time adaptation in river. *ACM Trans. Comput. Syst.* 21, 1 (2003), 36–86. https://doi.org/10.1145/592637.592639

[9] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*. 261–272. https://doi.org/10.1145/342009.335420

[10] Shivnath Babu and Pedro Bizarro. 2005. Adaptive Query Processing in the Looking Glass. In *CIDR*. 238–249. http://cidrdb.org/cidr2005/papers/P20.pdf

[11] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. 2005. Proactive Re-optimization. In *SIGMOD*. 107–118. https://doi.org/10.1145/1066157.1066171

[12] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive Ordering of Pipelined Stream Filters. In *SIGMOD*. 407–418. https://doi.org/10.1145/1007568.1007615

[13] Shivnath Babu and Jennifer Widom. 2004. StreaMon: An Adaptive Engine for Stream Query Processing. In *SIGMOD*. https://doi.org/10.1145/1007568.1007702

[14] Henriette Behr, Volker Markl, and Zoi Kaoudi. 2023. Learn What Really Matters: A Learning-to-Rank Approach for ML-based Query Optimization. In *BTW*. 535–554. https://doi.org/10.18420/BTW2023-25

[15] Kevin S. Beyer, Peter J. Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. 2007. On synopses for distinct-value estimation under multiset operations. In *SIGMOD*. 199–210. https://doi.org/10.1145/1247480.1247504

[16] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. 2005. Content-Based Routing: Different Plans for Different Data. In *VLDB*. http://www.vldb.org/archives/website/2005/program/paper/thu/p757-bizarro.pdf

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

[18] Matthias Boehm. 2011. *Cost-based optimization of integration flows.* Ph. D. Dissertation. Dresden University of Technology. https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-67936

[19] Matthias Boehm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. 2014. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.* 37, 3 (2014), 52–62. http://sites.computer.org/debull/A14sept/p52.pdf

[20] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H.. In *TPCTC*, Raghunath Nambiar and Meikel Poess (Eds.), Vol. 10661. 103–119. http://dblp.uni-trier.de/db/conf/tpctc/tpctc2017.html#BonczAK17

[21] Renata Borovica-Gajic, Goetz Graefe, and Allison W. Lee. 2017. Robust Performance in Database Query Processing (Dagstuhl Seminar 17222). *Dagstuhl Reports* 7, 5 (2017), 169–180. https://doi.org/10.4230/DagRep.7.5.169

[22] Renata Borovica-Gajic, Goetz Graefe, Allison W. Lee, Caetano Sauer, and Pinar Tözün. 2022. Database Indexing and Query Processing (Dagstuhl Seminar 22111). *Dagstuhl Reports* 12, 3 (2022), 82–96. https://doi.org/10.4230/DagRep.12.3.82

[23] Nicolas Bruno and Surajit Chaudhuri. 2002. Exploiting statistics on query expressions for optimization. In *SIGMOD*. 263–274. https://doi.org/10.1145/564691.564722

[24] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*. http://www-db.cs.wisc.edu/cidr/cidr2003/program/p24.pdf

[25] Chung-Min Chen and Nick Roussopoulos. 1994. Adaptive Selectivity Estimation Using Query Feedback. In *SIGMOD*. https://doi.org/10.1145/191839.191874

[26] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*. 379–390. https://doi.org/10.1145/342009.335432

[27] Benoît Dageville et al. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[28] Amol Deshpande. 2004. An initial study of overheads of eddies. *SIGMOD Rec.* 33, 1 (2004), 44–49. https://doi.org/10.1145/974121.974129

[29] Amol Deshpande, Joseph M. Hellerstein, and Vijayshankar Raman. 2006. Adaptive query processing: why, how, when, what next. In *SIGMOD*. 806–807. https://doi.org/10.1145/1142473.1142603

[30] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends Databases* 1, 1 (2007), 1–140. https://doi.org/10.1561/1900000001

[31] Bailu Ding, Surajit Chaudhuri, and Vivek R. Narasayya. 2020. Bitvector-aware Query Optimization for Decision Support Queries. In *SIGMOD*. ACM, 2011–2026. https://doi.org/10.1145/3318464.3389769

[32] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. 2007. On the Production of Anorexic Plan Diagrams. In *VLDB*. 1081–1092. http://www.vldb.org/conf/2007/papers/research/p1081-d.pdf

[33] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. 2008. Identifying robust plans through plan diagram reduction. *PVLDB* 1, 1 (2008), 1124–1140. https://doi.org/10.14778/1453856.1453976

[34] Anshuman Dutt and Jayant R. Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). 1039–1050. https://doi.org/10.1145/2588555.2588566

[35] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *PVLDB* 12, 9 (2019), 1044–1057. https://doi.org/10.14778/3329772.3329780

[36] Tom Ebergen. 2022. *Join Order Optimization with (Almost) No Statistics.* Master's thesis. https://homepages.cwi.nl/~boncz/msc/2022-TomEbergen.pdf

[37] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. *PVLDB* 10, 12 (2017), 1825–1836. https://doi.org/10.14778/3137765.3137786

[38] Goetz Graefe, Wey Guy, Harumi A. Kuno, and Glenn N. Paulley. 2012. Robust Query Processing (Dagstuhl Seminar 12321). *Dagstuhl Reports* 2, 8 (2012), 1–15. https://doi.org/10.4230/DagRep.2.8.1

[39] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *SIGMOD*. 2487–2503. https://doi.org/10.1145/3318464.3389739

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

[41] Immanuel Haffner and Jens Dittrich. 2023. Efficiently Computing Join Orders with Heuristic Search. *Proc. ACM Manag. Data* 1, 1 (2023), 73:1–73:26. https://doi.org/10.1145/3588927

[42] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. 2008. Parallelizing query optimization. *PVLDB* 1, 1 (2008), 188–200. https://doi.org/10.14778/1453856.1453882

[43] Jayant R. Haritsa. 2010. The Picasso Database Query Optimizer Visualizer. *PVLDB* 3, 2 (2010), 1517–1520. https://doi.org/10.14778/1920841.1921027

[44] Jayant R. Haritsa. 2020. Robust Query Processing: Mission Possible. *PVLDB* 13, 12 (2020), 3425–3428. https://doi.org/10.14778/3415478.3415561

[45] Benjamin Hilprecht and Carsten Binnig. 2022. One Model to Rule them All: Towards Zero-Shot Learning for Databases. In *CIDR*. https://www.cidrdb.org/cidr2022/papers/p16-hilprecht.pdf

[46] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *PVLDB* 15, 11 (2022), 2361–2374. https://www.vldb.org/pvldb/vol15/p2361-hilprecht.pdf

[47] Urs Hölzle and David M. Ungar. 1994. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *PLDI*. 326–336. https://doi.org/10.1145/178243.178478

[48] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the Black Boxes in Data Flow Optimization. *PVLDB* 5, 11 (2012), 1256–1267. https://doi.org/10.14778/2350229.2350244

[49] IBM. 2005. An architectural blueprint for autonomic computing. Whitepaper.

[50] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *SIGMOD*. 647–658. https://doi.org/10.1145/1007568.1007641

[51] Yannis E. Ioannidis. 1993. Universality of Serial Histograms. In *VLDB*. 256–267. http://www.vldb.org/conf/1993/P256.PDF

[52] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*. 268–277. https://doi.org/10.1145/

115790.115835
[53] Zachary G. Ives, Amol Deshpande, and Vijayshankar Raman. 2007. Adaptive query processing: Why, How, When, and What Next?. In *VLDB*. 1426–1427. http://www.vldb.org/conf/2007/papers/tutorials/p1426-deshpande.pdf
[54] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. 2004. Adapting to Source Properties in Processing Data Integration Queries. In *SIGMOD*. 395–406. https://doi.org/10.1145/1007568.1007613
[55] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. 2021. COMPASS: Online Sketch-based Query Optimization for In-Memory Databases. In *SIGMOD*. 804–816. https://doi.org/10.1145/3448016.3452840
[56] Vanja Josifovski, Peter M. Schwarz, Laura M. Haas, and Eileen Tien Lin. 2002. Garlic: a new flavor of federated query processing for DB2. In *SIGMOD*. 524–532. https://doi.org/10.1145/564691.564751
[57] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD*. 106–117. https://doi.org/10.1145/276304.276315
[58] Carl-Christian Kanne and Guido Moerkotte. 2010. Histograms reloaded: the merits of bucket diversity. In *SIGMOD*. 663–674. https://doi.org/10.1145/1807167.1807239
[59] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *PVLDB* 7, 12 (2014), 1119–1130. https://doi.org/10.14778/2732977.2732986
[60] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf
[61] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504. https://doi.org/10.1145/3183713.3196909
[62] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *SIGMOD*. 239–250. https://doi.org/10.1145/2723372.2742788
[63] Kukjin Lee, Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2023. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. *PVLDB* 16, 11 (2023), 2871–2883. https://doi.org/10.14778/3611479.3611494
[64] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594
[65] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594
[66] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. 2007. Adaptively Reordering Joins during Query Execution. In *ICDE*. 26–35. https://doi.org/10.1109/ICDE.2007.367848
[67] Guy M. Lohman. 2017. Query Optimization - Are We There Yet?. In *BTW*. 25–26. https://dl.gi.de/handle/20.500.12116/646
[68] Ryan Marcus. 2023. Learned Query Superoptimization. *CoRR* abs/2303.15308 (2023). https://doi.org/10.48550/arXiv.2303.15308
[69] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. 1275–1288. https://doi.org/10.1145/3448016.3452838
[70] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644
[71] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. 2004. Robust Query Processing through Progressive Optimization. In *SIGMOD*. 659–670. https://doi.org/10.1145/1007568.1007642
[72] Guido Moerkotte. 2023. Building Query Compilers. https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf Last Accessed: February 9, 2024.
[73] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*. 930–941.
[74] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *SIGMOD*. 539–552. https://doi.org/10.1145/1376616.1376672
[75] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *PVLDB* 4, 11 (2011), 843–851. http://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf
[76] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB* 2, 1 (2009), 982–993. https://doi.org/10.14778/1687627.1687738
[77] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. In *MLSys*. https://proceedings.mlsys.org/book/272.pdf
[78] Magnus Müller and Guido Moerkotte. 2022. Translation Grids for Multi-way Join Size Estimation. In *EDBT*. 2:378–2:382. https://doi.org/10.48786/edbt.2022.25
[79] P E O'Neil, E J O'Neil, and X Chen. 2009. The Star Schema Benchmark (SSB). https://cs.umb.edu/~poneil/StarSchemaB.pdf Last Accessed: February 9, 2024.
[80] Neoklis Polyzotis. 2005. Selectivity-based partitioning: a divide-and-union paradigm for effective query optimization. In *CIKM*. 720–727. https://doi.org/10.1145/1099554.1099730
[81] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*. 1981–1984. https://doi.org/10.1145/3299869.3320212
[82] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. 2013. Micro adaptivity in Vectorwise. In *SIGMOD*. 1231–1242. https://doi.org/10.1145/2463676.2465292
[83] Naveen Reddy and Jayant R. Haritsa. 2005. Analyzing Plan Diagrams of Database Query Optimizers. In *VLDB*. 1228–1240. http://www.vldb.org/archives/website/2005/program/paper/fri/p1228-reddy.pdf
[84] Alice Rey, Michael Freitag, and Thomas Neumann. 2023. Seamless Integration of Parquet Files into Data Processing. In *BTW*. 235–258. https://doi.org/10.18420/BTW2023-12
[85] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2023. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 2 (2023), 11:1–11:38. https://doi.org/10.1145/3485126
[86] Nils L. Schubert, Philipp M. Grulich, Steffen Zeuch, and Volker Markl. 2023. Exploiting Access Pattern Characteristics for Join Reordering. In *DaMoN@SIGMOD*. 10–18. https://doi.org/10.1145/3592980.3595304
[87] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. 23–34. https://doi.org/10.1145/582095.582099
[88] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-architectural Analysis of In-memory OLTP. In *SIGMOD*. 387–402. https://doi.org/10.1145/2882903.2882916
[89] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB*. 19–28. http://www.vldb.org/conf/2001/P019.pdf
[90] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD*, Carlo Zaniolo (Ed.). ACM Press, 340–355. https://doi.org/10.1145/16894.16888
[91] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load Shedding in a Data Stream Manager. In *VLDB*. 309–320. https://doi.org/10.1016/B978-012722442-8/50035-5
[92] Transaction Processing Council. 1993. TPC Benchmark H (Decision Support). https://www.tpc.org/tpch/ Last Accessed: February 9, 2024.
[93] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*. 1153–1170. https://doi.org/10.1145/3299869.3300088
[94] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *ACM Trans. Database Syst.* 46, 3 (2021), 9:1–9:45. https://doi.org/10.1145/3464389
[95] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. 2010. Sharing-Aware Horizontal Partitioning for Exploiting Correlations During Query Processing. *PVLDB* 3, 1 (2010), 542–553. https://doi.org/10.14778/1920841.1920911
[96] Li Wang, Tom Z. J. Fu, Richard T. B. Ma, Marianne Winslett, and Zhenjie Zhang. 2019. Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing. In *SIGMOD*. 573–588. https://doi.org/10.1145/3299869.3319868
[97] Ziyun Wei and Immanuel Trummer. 2022. SkinnerMT: Parallelizing for Efficiency and Robustness in Adaptive Query Processing on Multicore Platforms. *PVLDB* 16, 4 (2022), 905–917. https://www.vldb.org/pvldb/vol16/p905-wei.pdf
[98] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *PVLDB* 13, 3 (2019), 279–292. https://doi.org/10.14778/3368289.3368294
[99] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*. 423–438. https://doi.org/10.1145/2517349.2522737
[100] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *CIDR*. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf
[101] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. 2016. Non-Invasive Progressive Optimization for In-Memory Databases. *PVLDB* 9, 14 (2016), 1659–1670. https://doi.org/10.14778/3007328.3007332
[102] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust. *PVLDB* 10, 8 (2017), 889–900. https://doi.org/10.14778/3090163.3090167