Scalable Multi-Query Execution using Reinforcement Learning

Panagiotis Sioulas
EPFL
Lausanne, Switzerland
panagiotis.sioulas@epfl.ch

Anastasia Ailamaki
EPFL, RAW Labs SA
Lausanne, Switzerland
anastasia.ailamaki@epfl.ch

ABSTRACT

The growing demand for data-intensive decision support and the migration to multi-tenant infrastructures put databases under the stress of high analytical query load. The requirement for high throughput contradicts the traditional design of query-at-a-time databases that optimize queries for efficient serial execution. Sharing work across queries presents an opportunity to reduce the total cost of processing and therefore improve throughput with increasing query load. Systems can share work either by assessing all opportunities and restructuring batches of queries ahead of execution, or by inspecting opportunities in individual incoming queries at runtime: the former strategy scales poorly to large query counts, as it requires expensive sharing-aware optimization, whereas the latter detects only a subset of the opportunities. Both strategies fail to minimize the cost of processing for large and ad-hoc workloads.

This paper presents RouLette, a specialized intelligent engine for multi-query execution that addresses, through runtime adaptation, the shortcomings of existing work-sharing strategies. RouLette scales by replacing sharing-aware optimization with adaptive query processing, and it chooses opportunities to explore and exploit by using reinforcement learning. RouLette also includes optimizations that reduce the adaptation overhead. RouLette increases throughput by 1.6-28.3x, compared to a state-of-the-art query-at-a-time engine, and up to 6.5x, compared to sharing-enabled prototypes, for multi-query workloads based on the schema of TPC-DS.

KEYWORDS

sharing, multi-query optimization, join, reinforcement learning

ACM Reference Format:
Panagiotis Sioulas and Anastasia Ailamaki. 2021. Scalable Multi-Query Execution using Reinforcement Learning. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452799

1 INTRODUCTION

The growing need for data-driven decision-making and the increasing prevalence of multi-tenant infrastructures significantly increases the analytical query load [8, 17]. Traditionally, state-of-the-art analytical engines follow a query-at-a-time execution model and

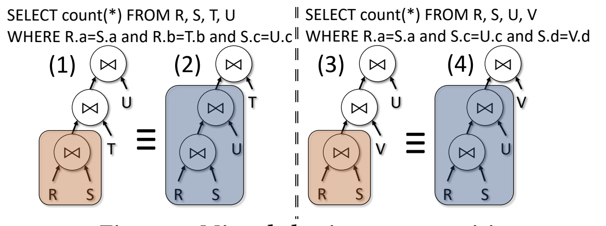

SELECT count(*) FROM R, S, T, U
WHERE R.a=S.a and R.b=T.b and S.c=U.c
SELECT count(*) FROM R, S, U, V
WHERE R.a=S.a and S.c=U.c and S.d=V.d
(1)
(2)
(3)
(4)
⋈
R
S
T
U
V
≡

Figure 1: Missed sharing opportunities

are a poor fit for high query-load scenarios. Throughput-oriented DBMS [7, 13, 16, 17] handle multi-query processing more efficiently by taking advantage of shared data and work across queries to reduce the amount of work to perform. Still, there is no silver bullet for finding the sharing decisions that minimize the cost of processing.

Work-sharing is either online [2, 7, 16] or offline [14, 33, 43]. *Online sharing* detects opportunities, such as common sub-expressions, between incoming and ongoing queries at runtime. Although the detection overhead is low (e.g., matching sub-expressions), online sharing finds only a subset of the opportunities in the workload. Figure 1 demonstrates this limitation. To minimize the cost of processing individual queries, query optimization produces query plans (1) and (3). The plans share the first join, $R \bowtie S$. However, there exist equivalent plans (2) and (4) with permuted join orders that can share $R \bowtie S \bowtie U$, thus reducing the total cost. The permutation constitutes a missed opportunity for online sharing. *Offline sharing* optimizes batches of queries, by using sharing-aware optimization, to form a global query plan that minimizes the total cost of query processing. Offline sharing discovers opportunities that online sharing misses. However, sharing-aware optimization is a high complexity problem that takes several seconds to process batches as small as few tens of queries [14]. As it lies in the critical path of execution, it obstructs offline sharing to scale to hundreds of queries, especially in ad-hoc workloads. Therefore, depending on the use or absence of sharing-aware optimization, existing systems either forfeit support for large-scale workloads or miss substantial sharing opportunities.

We preserve scalability and maximize exploited opportunities. Scalability requires avoiding the cost of sharing-aware optimization being paid in full. This requirement contradicts the *optimize-then-execute* paradigm that most DBMS adopt. A continuously adaptive paradigm that, by using heuristics, re-optimizes queries at runtime is a better fit for large workloads, as it moves optimization out of the critical path by permitting execution to proceed alongside plan refinement. To maximize the benefit of sharing, intelligent heuristics can steer exploratory decisions toward efficient global query plans, by monitoring execution outcomes.

In this paper, we present RouLette, a novel intelligent engine that exposes and exploits shareable work among Select-Project-Join (SPJ) sub-queries, through runtime adaptation. RouLette operates

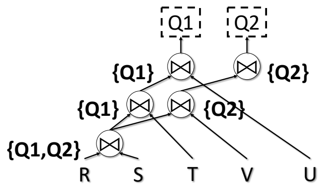


Figure 2: Global Query Plan

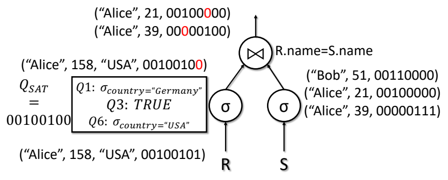


Figure 3: Shared Operators

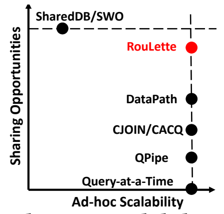


Figure 4: Sharing-Scalability tradeoff

in fine-grained episodes. During each episode, it performs work for all of the ongoing queries, monitors the cardinalities of intermediate results, and adjusts the plan by using a learned heuristic. The learned heuristic estimates, by using reinforcement learning, which planning decisions minimize the total cost, hence steering adaptation toward more efficient plans compared to online sharing mechanisms. By continuously adapting the global query plan, RouLette minimizes work while preserving scalability.

Contributions: We make the following contributions:

- We present a work-sharing paradigm that both minimizes the total cost and overcomes the scalability limitation of sharing-aware optimization. By using adaptive processing, RouLette explores and exploits opportunities at runtime hence addresses the drawbacks of online and offline sharing.
- We design a sharing-aware heuristic that overcomes, by using reinforcement learning, the limitations of existing heuristics [4, 38] for runtime planning and produces efficient global plans. For batches of Join Order Benchmark queries, it produces, on average, 3.2x fewer intermediate tuples than a greedy selectivity-based heuristic and 1.4x fewer intermediate tuples than sharing-oblivious learned heuristics.
- We identify performance bottlenecks inherited by (i) adaptive processing, that is materializing join state and intermediate tuples, and (ii) shared operators, that is filter comparisons and routing. We propose novel optimizations, i.e., symmetric join pruning, adaptive projections, range-based grouped filters and locality-conscious routers, that improve hardware utilization and scalability. With optimizations, RouLette increases throughput, compared to state-of-the-art DBMS, by 1.6-28.3x and, compared to online sharing, by up to 6.5x.

2 BACKGROUND & RELATED WORK

We first provide an overview of the areas RouLette builds upon.

2.1 Work-Sharing

In this section, we present the two core concepts of work-sharing, global query plans and the Data-Query model, and provide an overview of existing online and offline sharing systems.

Global Query Plan: Work-sharing exploits matching sub-expressions across queries. By overlapping plans, shared operators form a DAG, called the *global query plan*, that processes multiple queries at once. Shared operators route their output to one or more parent operators that are often shared. Figure 2 shows the DAG for queries (1) and (3) from Figure 1. A shared operator processes $R \bowtie S$ for both queries, then routes results to a different parent for each query.

Data-Query Model: Exact sub-expression matching limits opportunities due to diverse selections. By augmenting tuples with query sets, the *Data-Query model* [2, 7, 13, 19, 24] enables queries to share sub-expressions with the same join order and different selections. The model expresses tuples as $a = (a_1, a_2, \ldots, a_n, \overline{a_q})$ where $a_1, a_2, \ldots, a_n$ are attributes and a_q is the set of queries a belongs to.

Data-Query operators form global plans that process augmented tuples. We present shared selections and joins, depicted in Figure 3. *Shared Selection:* It evaluates at least one predicate (or a TRUE predicate, if there is none) per query. An input tuple a satisfies the selection's predicates for queries $Q_{sat}(a)$. To exclude queries with false predicates, selection updates a_q to $a_q \cap Q_{sat}(a)$.
Shared Join: It matches Data-Query model tuples from its inputs. For each match, the join produces a new shared tuple that belongs to the intersection of the query-sets of the matching tuples.

Taxonomy of Sharing: We classify shared-work systems based on the mechanism used for detecting opportunities: online sharing, which detects opportunities at runtime, and offline sharing, which uses sharing-aware optimization. Figure 4 evaluates existing systems on their ability to (i) exploit opportunities and (ii) scale with the number of ad-hoc queries. Systems using online sharing can scale to large ad-hoc workloads, whereas systems using offline sharing maximize exploitation.

Online sharing makes only locally-optimal sharing decisions. QPipe [16] and DataPath [2] detect opportunities at query-level. QPipe shares only common sub-plans. DataPath extends a global query plan to incorporate incoming queries with minimum additional cost hence it is sensitive to the admission order. CJOIN [7] and CACQ [19, 24] detect opportunities at operator-level. They both reorder operators at runtime based on selectivity, hence they miss operator correlations and the long-term effects of planning. In all cases, online sharing misses opportunities.

Offline sharing uses sharing-aware optimization to find a minimum-cost global plan. Early Multi-query Optimization (MQO) algorithms [30, 34, 35] exhaustively explore a doubly exponential space in the batch size hence are expensive. More recent algorithms [33, 43] can optimize tens of queries at once. Shared-workload Optimizers, such as SWO [14], produce plans made of Data-Query operators. Sharing-aware optimization targets recurring workloads with few queries and is a poor fit for large and ad-hoc workloads.

Shared-work systems need to execute global plans efficiently. Materialized views [33, 43] and pipelining [9] are two options. SharedDB [13] introduces the batched execution model that maximizes sharing for a wider range of operators and queries. MQJoin [25] enhances the batched model with a high-throughput join. Both SharedDB and MQJoin depend on using SWO to produce a global query plan and hence cannot scale to large workloads.

Online sharing misses opportunities, whereas offline sharing depends on sharing-aware optimization, which has very high complexity and restricts scalability. RouLette overcomes the scalability limitation by replacing optimization with adaptation. Also, by reordering operators using a learned heuristic, it overcomes the limitations of query-local and selectivity-based decisions. Hence, RouLette can maximize throughput for large ad-hoc workloads.

2.2 Adaptive Query Processing

Adaptive processing targets use cases where the optimize-then-execute paradigm performs poorly e.g. unpredictable environments with limited statistics and highly correlated data. It adapts planning during execution by exploiting information collected at runtime. In this section, we present the adaptive techniques that RouLette uses: symmetric hash joins, eddies, and State Modules.

Symmetric Hash-join (SHJ): Hash-joins limit opportunities for runtime adaptation, as the choice of build relations and the execution order is static. By treating inputs equally, *SHJ* [15, 40] processes tuples from both inputs in any interleaved order. SHJ builds hashtables on both inputs. It inserts every input tuple in the respective hashtable, then probes the other relation's hashtable for matches. SHJ produces each result tuple when the matching tuples from both sides have been consumed. Figure 5a shows an example for matching two tuples. SHJ processes R's tuple with key 4, inserts it in R's hashtable, and, without matches, probes S. Next, SHJ processes S's tuple, also with key 4, inserts it in S's hashtable, and, to match with R's preceding tuple, probes R. Thus, SHJ enables out-of-order processing but increases materialization cost and footprint.

SHJ generalizes to n-ary joins. It probes n-1 hashtables, deciding the order at runtime. N-ary SHJ is popular in stream processing [11, 37] and robust query processing [6, 21].

Eddies: An *eddy* operator [3] reorders operators in plans at runtime. Specifically, it controls how tuples flow through operators. By observing the input and output of operators, it optimizes the operator order. Thus, the eddy replaces the optimizer with adaptation.

Eddies can adapt plans with commutative and symmetric operators, such as SHJ. However, accumulated operator state limits adaptability e.g. inserted tuples in SHJ cannot include more joins until they are probed. State makes routing history-dependent [10].

State Modules (STeMs): *STeMs* [32] enhance the adaptability of eddies. A STeM is an index that stores tuples for each base relation. It exposes two operations, *insert(a)* and *probe(a)*. Insert stores tuple a in the STeM and probe joins, based on a key, with previously inserted tuples. STeMs store tuples at endpoints and avoid materializing intermediate tuples, hence guarantee history-independence.

STeMs can implement n-ary SHJ [32]. Figure 5b shows a 3-way SHJ. STeMs serve as hashtables, whereas the eddy dynamically reorders probes. The eddy first inserts each tuple to its relation's STeM, e.g., R, producing an insertion timestamp. Then, it atomically probes other STeMs e.g. $STeM_S$ then $STeM_T$, using the timestamp to ensure atomicity. Probe sequences produce the output tuples.

RouLette replaces sharing-aware optimization with STeM-based adaptation. As the eddy produces global plans using fast decisions, its cost is linear to the plans' size hence scalable. RouLette enhances adaptive processing with efficient planning and reduces overhead.

2.3 Reinforcement Learning

To emulate query optimization, adaptive processing requires efficient operator orders. Existing selectivity-based techniques are greedy hence often sub-optimal. To refine ordering and to increase sharing benefits, RouLette uses reinforcement learning [36].

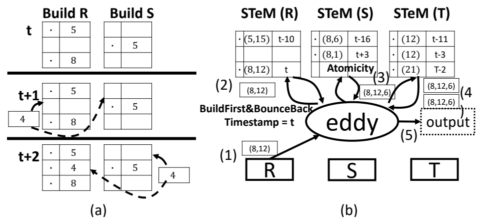


Figure 5: (a) SHJ (b) 3-way SHJ with STeMs

Reinforcement learning applies to problems that are expressed as Markov Decision Processes (MDPs). An MDP models problems as multi-step processes. At each step, the actor observes the current state s and chooses an action $a \in A(s)$. The action's result is a reward $R(s, a)$ and a change of state to s'. The state space, each state's set of actions, the reward, and state transitions define the MDP. Eventually, the actor observes a terminal state and the process finishes. Reinforcement learning algorithms find a decision-making policy that maximizes the cumulative reward that the agent observes.

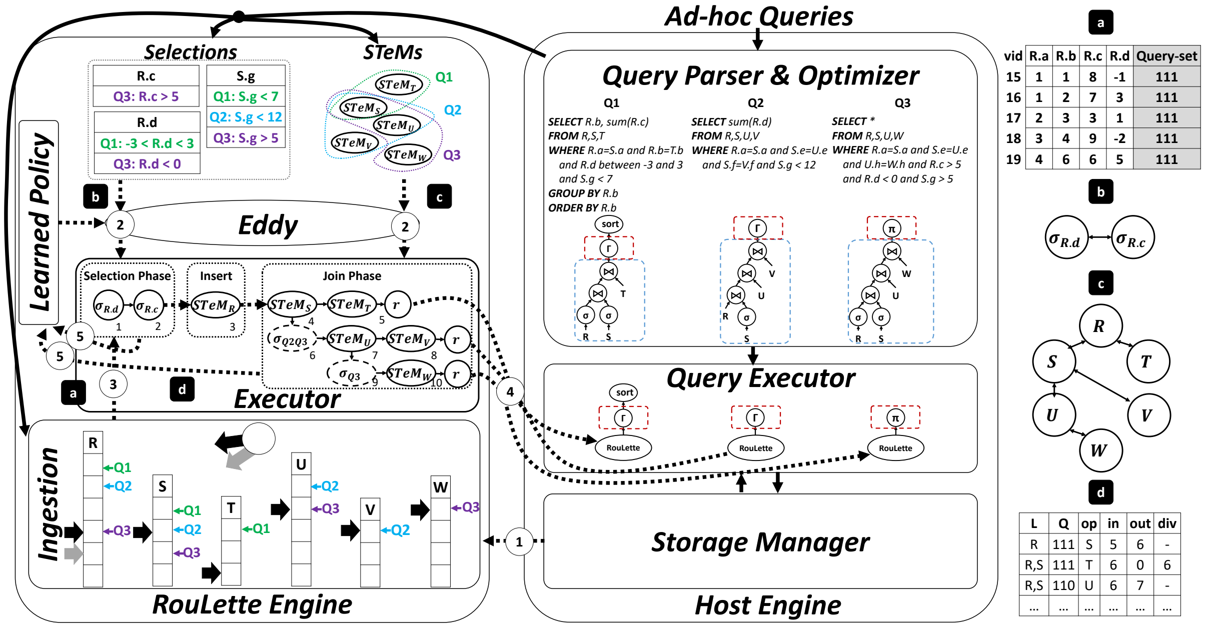
Q-learning [39] is a reinforcement learning algorithm for finding optimal policies. RouLette uses Q-learning because it has multiple desirable properties: (1) it learns the optimal policy instead of the currently used policy, i.e. a randomized policy that explores opportunities. (2) the only convergence requirement is that all state-action pairs are updated, which is guaranteed with randomized decisions.

Q-learning approximates a function $Q : S \times A \longrightarrow \mathbb{R}$ that evaluates the quality of decision a at state s. During decision-making, the policy chooses, with probability 1-ϵ, the action a that maximizes $Q(s, a)$ and a random action otherwise. Q-learning later uses the observed rewards to refine the approximation of Q. Given two hyper-parameters, learning rate μ and discount rate γ, it updates: $Q(s, a) \leftarrow Q(s, a) + \mu(r_t + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a))$.

Recent work uses reinforcement learning to build query-at-a-time learned query optimizers [20, 26, 27, 42]. Learned optimizers are trained offline and improve planning throughout a sequence of queries. By contrast, RouLette learns the ordering heuristic throughout the lifetime of queries. Learning is completely online and discards information after queries finish processing. We make this design choice for two reasons: (i) predictions for a batch do not generalize for seemingly similar future batches, because they depend on the batch's predicates, which are rarely the same, and (ii) the exact same batch recurs less often than the exact same sub-query.

2.4 Learned Cardinality Estimation

Cardinality and selectivity estimates are often inaccurate [22] and cause query optimizers to choose suboptimal plans. To achieve low-error estimates within tight latency and storage constraints, recent work proposes novel machine learning techniques, such as cost-guided cardinality estimation [29], progressive sampling over autoregressive models [41], MSCNs [18], and domain-specific feature and label engineering [12]. Similar to learned optimizers, learned cardinality estimation is trained offline. RouLette sidesteps cardinality estimation, as it can measure cardinalities at runtime.



Q1
```
SELECT R.b, sum(R.c)
FROM R,S,T
WHERE R.a=S.a and R.b=T.b
   and R.d between -3 and 3
   and S.g < 7
GROUP BY R.b
ORDER BY R.b
```

Q2
```
SELECT sum(R.d)
FROM R,S,U,V
WHERE R.a=S.a and S.e=U.e
   and S.f=V.f and S.g < 12
```

Q3
```
SELECT *
FROM R,S,U,W
WHERE R.a=S.a and S.e=U.e
   and U.h=W.h and R.c > 5
   and R.d < 0 and S.g > 5
```

a

vid	R.a	R.b	R.c	R.d	Query-set
15	1	1	8	-1	111
16	1	2	7	3	111
17	2	3	3	1	111
18	3	4	9	-2	111
19	4	6	6	5	111

d

L	Q	op	in	out	div
R	111	S	5	6	-
R,S	111	T	6	0	6
R,S	110	U	6	7	-
...	...	...	...	...	...

Figure 6: RouLette operates as a special engine alongside a DBMS. We present the architecture through a three-query example.

3 ROULETTE ARCHITECTURE

We introduce RouLette, a specialized intelligent engine for efficiently executing multiple SPJ sub-queries at once. By continuously adapting the global plan to sharing opportunities, it maximizes sub-query processing throughput. Thus, RouLette optimizes the global operator order, without expensive sharing-aware optimization.

Figure 6 shows RouLette's architecture. A host DBMS processes concurrent queries from different users and applications. The host delegates SPJ sub-queries to RouLette for high-throughput processing and, then, collects the results for further processing. RouLette works separately alongside the host because (i) it uses a different processing paradigm (adaptive instead of optimize-then-execute), (ii) it processes work across and beyond the lifetime of queries using a global instance, and (iii) it controls its input, state, and execution.

RouLette splits sub-query processing into *episodes*. In each episode, RouLette plans then processes the operators of ongoing sub-queries for an input vector and analyzes execution to refine planning in future episodes. Episodes are the quantum of planning; plans change only across episodes. They process shared work for all ongoing sub-queries and map 1-1 to vectors (1024 input tuples in our prototype). Sub-queries finish after RouLette processes all their input.

Numbered dotted lines in Figure 6 show the data flow between RouLette's components in each episode. (1) *Ingestion* pulls a vector from the host's storage into RouLette. (2) An *eddy* within RouLette chooses the episode's plans for selections and joins using a *learned policy*. (3) The *executor* carries out, by processing the episode's plans for the vector, the eddy's decisions and produces SPJ results. (4) The executor pipelines results to host-side operators (e.g., GROUP BY, outer queries) for further processing. (5) The eddy uses execution metadata to refine the learned policy. We present RouLette's components using the example of the three queries in Figure 6.

Query Optimizer: The optimizer processes incoming queries and produces plans. Then, it delegates one or more SPJ sub-queries (blue boxes), which naturally occur at the bottom of plans, to RouLette. In the host, by replacing sub-queries with *RouLette sources*, delegation transforms the original plans and assigns the transformed plans to the host's executor. RouLette sources represent intra-RouLette processing and pipeline SPJ results to their consumer (i.e., parent) operator (red boxes). As RouLette does not preserve interesting orders, the optimizer, during transformation, also adds any required *sort* as a consumer. In RouLette, delegated sub-queries are scheduled, either online or in batches. Scheduling updates the predicate list and the join list, and notifies ingestion about new queries. After scheduling, RouLette starts processing the sub-queries. In the example, the host delegates Q1, Q2 and Q3 one after the other.

Ingestion: Ingestion provides RouLette with vectors from the host's storage. It is designed for two desirable properties: (i) to ensure that all ongoing queries make progress, and (ii) to enable sharing between incoming and ongoing queries in dynamic workloads. To satisfy (i) it scans relations in round-robin order, whereas to satisfy (ii) it uses circular scans [16, 44].

In each episode, ingestion chooses (i) a relation to access and (ii) the relation's vector to access. Hence, ingestion uses a relation iterator and a vector iterator for each relation. In the example, it chooses R, then R's 4th vector, and finally advances the two iterators. As scans are circular, retrieving the last vector of a relation (e.g., R's 6th) will move the iterator back to the start (e.g., R's 1st).

Ingestion also transforms input to Data-Query model. By recording the position of scans during each query's scheduling, ingestion tracks each scan's active queries i.e., queries that have not completed the circular scan. To translate the input to Data-Query model, it annotates tuples with the bitset of active queries. A set i^{th} bit means that the tuple belongs to Q_i e.g. if $Q1$, $Q2$ and $Q3$ are active, the tuple is annotated with 111. Figure 6a shows the resulting vector. When a query's circular scans are all finished, it becomes inactive, and hence ingestion signals the consumer with *end-of-input.*

STeMs: RouLette uses STeMs to enable operator reordering and out-of-order scans. They store and index tuples, making them accessible across episodes, without limiting future operator orders. Thus, RouLette implements a history-independent multi-query n-ary symmetric join. To address performance and parallelization bottlenecks in STeMs, RouLette introduces novel optimizations, i.e., symmetric join pruning, scalable versioning, and adaptive projections. We discuss STeM implementation in Sections 5.1 and 5.2.

Eddy & Learned Policy: The eddy handles planning within each episode and adaptation across episodes. It produces global plans that process all sub-queries for one episode each, and analyzes the plans' execution to produce more efficient plans in future episodes. Unlike prior work, RouLette uses adaptation for scaling sharing-aware optimization rather than for robustness. Also, it produces more efficient plans than existing adaptation techniques, as it uses novel learned policies that can accurately model costs.

RouLette uses selection push-down. As joins are much more costly, to reduce their input, plans first process the selections and then the joins. Hence, each episode has two separate plans, the selection-phase that processes shared selections and the join-phase that comprises STeM probes, routing selections and output routers. Figure 6 shows the two plans inside the executor.

To produce each phase's plan, the eddy chooses an operator order using a multi-step optimization algorithm. Multi-step optimization uses learned policies and ordering constraints (Figures 6b and 6c, presented in Section 4.1) to incrementally build the plan from unordered operators. We discuss optimization in Section 4.1.

The eddy continuously refines learned policies using reinforcement learning. By monitoring the input and output of operators, it collects an execution log that records the processed operator and queries, the operator history, and the size of input and output (Figure 6d). At the end of the episode, to update the policies and thus improve planning in future episodes, it processes the log using a tailor-made novel variant of Q-learning that reduces the problem's state space. We discuss learning in Sections 4.2 and 4.3.

Executor: The executor contains a pool of RouLette workers. Each worker concurrently undertakes a different episode and synchronizes with other workers through shared STeMs. It processes the episode's plans for the ingested vector mapped to the episode as follows. First, it processes the selection-phase, thus filtering the tuples' query-sets. Second, it inserts the selection phase's results to the base relation's STeM (e.g. $STeM_R$) to make the join symmetric. Third, it processes the join-phase for the selection-phase's results, to produce SPJ results. Fourth, using routers, it sends SPJ results to respective RouLette sources, which pipeline tuples to host operators. Each processed episode contributes to completing the sub-queries.

In all steps, RouLette's operators use the Data-Query model and serve one or more queries. RouLette introduces novel algorithms for efficiently processing selections and routing at scale, and, to maximize sharing, adopts MQJoin for STeM operations. We defer discussing operator implementation until Section 5.

The worker processes phases using vectorized execution. It processes each operator for an input vector and sends the output vector to one ($\sigma_{R.d}$'s case) or two operators ($STeM_S$'s case). When one

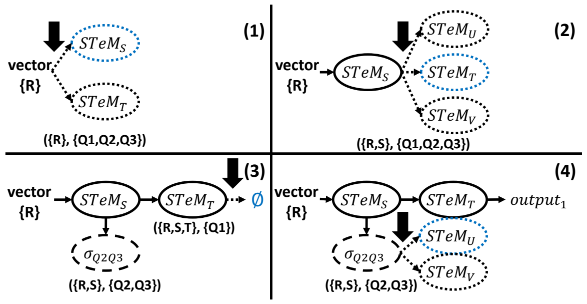


Figure 7: Multi-step optimization for join-phase

operator follows, the worker executes it next. When two operators follow, to bound pending vector footprint, it executes all operators in the probe sub-plan first, then all operators in the selection sub-plan, e.g. after $STeM_S$ probe, data flow is (i) $STeM_S \longrightarrow STeM_T$, (ii) $STeM_T \longrightarrow Q1$'s RouLette source, (iii) $STeM_S \longrightarrow \sigma_{Q2,Q3}$, e.t.c. In Figure 6, numbers next to operators show the execution order.

Our prototype targets in-memory analytics by using columnar data and late materialization. Vectors consist of virtual ID (vID) tuples in PAX-layout [1], and operators reconstruct mini-columns for required attributes on demand. The design reduces the footprint of vectors and STeMs. As STeMs are in-memory, their footprint imposes an upper bound to the dataset size that RouLette can process.

4 LEARNED ADAPTATION POLICY

We examine planning in RouLette. We present how a policy produces global plans in Section 4.1, express planning as an MDP in Section 4.2, and propose a novel learning algorithm in Section 4.3.

4.1 Policy-based Planning

The eddy optimizes plans using policy decisions. Starting from the plan's input, each decision chooses the operators that process an intermediate vector. One or two chosen operators produce equally many vectors. Eventually, the plan contains all operators for all queries. The decision sequence is a *multi-step optimization.* It runs at each episode's start, and plans operators until the episode's end.

In this section, we present multi-step optimization. We follow the algorithm's first four steps for the running example in Figure 7.

Terminology: We first define the terms *dependency graph*, *lineage*, *operator query set*, *virtual vector*, and *candidate operator*,.

Definition 1. The **dependency graph** of a set of operators, O, is defined as the complete graph $K_{|O|}$ if O comprises selections and as $G = (V, E)$ with $(e_1, e_2) \in E$ iff $e_1 \bowtie e_2 \in O$ if O comprises joins.

Definition 2. Let O a set of operators with dependency graph $G(O)$. A subset $\mathcal{L} \subset O$ is defined as a **lineage** iff the induced subgraph $G(O)[\mathcal{L}]$ is connected. The set of lineages is $\mathcal{L}^*$.

Definition 3. The **query-set** Q_o of an operator o is defined as the set of queries that contain o.

Definition 4. A **virtual vector** is defined as a pair $(\mathcal{L}, Q)$.

Definition 5. **Candidate operators** for virtual vector $(\mathcal{L}, Q)$ are defined as $cand(\mathcal{L}, Q) = \{o \in O - \mathcal{L} | (\{o\} \cup \mathcal{L} \in \mathcal{L}^*) \wedge (Q \cap Q_o \neq \emptyset)\}$.

Definition 6. A **policy decision** for virtual vector $(\mathcal{L}, Q)$ is a function $\pi(\mathcal{L}, Q) = o$ with $o \in cand(\mathcal{L}, Q)$ if $cand(\mathcal{L}, Q) \neq \emptyset$, and $o = null$ otherwise.

Algorithm 1 Multi-step Optimization

```
1:  procedure MULTI_STEP_REC($node$, $\mathcal{L}$, $Q$)
2:      $next$ = NEXT_OPERATOR($\mathcal{L}, Q$)
3:      if $next \neq null$ then
4:          $main = node.addOperator(next, Q \cap Q_{next})$
5:          MULTI_STEP_REC($main$, $\mathcal{L} \cup \{next\}, Q \cap Q_{next}$)
6:          if $Q - Q_{next} \neq \emptyset$ then
7:              $div = node.addRoutingSelection(Q - Q_{next})$
8:              MULTI_STEP_REC($div$, $\mathcal{L}, Q - Q_{next}$)
9:      $node.addRouter(Q)$
10: procedure MULTI_STEP($relation, Q$)
11:     $input = InputNode(relation, Q)$
12:     MULTI_STEP_REC($input$, $\{relation\}, Q$)
13:     return $input$
```

Figures 6b and 6c show graphs for R's selections and joins. Dependency graphs express ordering constraints between operators: selections can execute in any order, whereas probes often need attributes from other relations to join without cross-products e.g. for $S.e = U.e$, R's tuples need to join with S before joining with U.

Virtual vectors (bold text Figure 7) represent shared sub-expressions that the eddy needs to expand, by adding operators, until they match one or more sub-queries. They contain a lineage, i.e., a set of operators that can compose a plan that respects constraints, and a query-set. $\{R, S\}$ is a lineage, while $\{R, U\}$ is a not lineage.

Candidates are operators that can extend the virtual vector's sub-expression. They need to respect constraints and the new sub-expression needs to be part of a query. Adding the operator to the lineage results in a new lineage. Dotted outlines (or $\emptyset$) highlight each step's candidates. In step (1), S and T are the only candidates because they are adjacent to R in the graph.

Policy decisions choose one of the candidates (blue outline), if any, as in steps (1), (2) and (4). If there are no candidates, as in step (3), the vector stands for Q's output and the policy returns *null*.

Next, we define the effects of decisions to partial global plans.
Sharing: Sharing occurs when all queries of virtual vector $(\mathcal{L}, Q)$ contain the operator o chosen by the policy, as in the example's step (1). The eddy shares o across Q and a new virtual vector $(\mathcal{L} \cup \{o\}, Q)$ stands for the new shared sub-expression.
Divergence: Divergence occurs when only a subset of Q contains o. Then, the eddy shares o only across that subset, $Q \cap Q_o$. Also, it shares a selection across the other queries, $Q - Q_o$, to drop redundant tuples. Hence, the decision results in two shared sub-expressions with virtual vectors $(\mathcal{L} \cup \{o\}, Q \cap Q_o)$ and $(\mathcal{L}, Q - Q_o)$. o shares work eagerly, as it processes all queries that contain Q_o. Step (2)'s decision causes divergence. As only $Q1$ contains $R \bowtie T$, the decision creates different sub-expressions for $Q1$ and $Q2$-$Q3$. Step (3) shows the resulting virtual vectors. Divergence routes sub-expressions to two outputs. To model more than two outputs, the eddy can make decisions that cause divergence consecutively.
Multi-step Optimization: To build a complete and correct global plan (i.e., implements delegated sub-queries), the eddy composes a sequence of inter-dependent policy decisions. We design multi-step optimization, the eddy's logic, which uses the policy to build the plan operator by operator and to identify the next decisions to make. Multi-step optimization is applied independently for the two phases, selection and join, to produce the two plans.

Algorithm 1 presents pseudocode for multi-step optimization. The algorithm recursively builds the global plan. At each recursive step, starting from the plan's input (lines 11-12), it chooses operators to add after the last operator of a shared sub-expression's plan. By using the policy, it first chooses a candidate of the sub-expression's virtual vector, o (line 2) and appends it to the plan for $Q \cap Q_o$ (line 4). Also, in case of divergence, it appends a selection for $Q - Q_o$ (line 7). The new operators' output corresponds to new sub-expressions. Multi-step optimization uses recursion to complete the plans of new sub-expressions for $Q \cap Q_o$ (line 5) and $Q - Q_o$ (line 8). Finally, *null* decisions indicate that the sub-expression is its query-set's output and a router to the host is added (line 9) When recursion finishes, the plan is complete.

We discuss Algorithm 1's correctness, complexity and optimality.
Correctness: MULTI_STEP_REC ($node$, $\mathcal{L}, Q$) produces a *null* decision, hence the query's output, for each $q \in Q$ exactly once.

Let $O_Q = \{o \in O | Q_o \cap Q \neq \emptyset\}$. We use induction on $|O_Q - \mathcal{L}|$. *Base step*: As $\mathcal{L} \subset O_Q$, $|O_Q - \mathcal{L}| = 0$ entails $\mathcal{L} = O_Q$. Then, $cand(\mathcal{L}, Q) = \emptyset$ hence the policy decides null once for each $q \in Q$. Induction step: If the proposition holds for $|O_Q - \mathcal{L}| \leq n$, it also holds for $|O_Q - \mathcal{L}| = n + 1$.

Let $o = \pi(\mathcal{L}, Q)$. $o \in O_{Q \cap Q_o}$ and $O_Q \subset O_{Q \cap Q_o}$ hence $|O_{Q \cap Q_o} - (\mathcal{L} \cup \{o\})| \leq n$. Recursion for $(\mathcal{L} \cup \{o\}, Q \cap Q_o)$ decides null exactly once for each $q \in (Q \cap Q_o)$.

If there is divergence, $o \notin O_{Q - Q_o}$ hence $|O_{Q - Q_o}| < O_Q$ and $|O_{Q - Q_o} - \mathcal{L}| \leq n$. Recursion for $(\mathcal{L}, Q - Q_o)$ decides null exactly once for each $q \in (Q - Q_o)$. The two recursions produce null for $(Q - Q_o) \cup (Q \cap Q_o) = Q$ and their query-sets do not overlap.
Complexity: The number of decisions is the global plan's size, which has at most Q_o instances of each operator o. Each decision inspects the candidates which are at most $|O|$. Hence, the worst-case complexity of Algorithm 1 is $O(|O| * \sum_{o \in O} |Q_o|)$.
Optimality: Algorithm 1 is optimal iff, at each step, the policy chooses the candidate that leads to the best possible plan given the decisions already made. Given an accurate estimate of each best possible plan's response time, it suffices to choose the candidate with the minimum estimate. The next section focuses on estimation.

4.2 Learning Policy Decisions

The response time of global plans depends on decision quality. To improve decision quality, the eddy adapts the policy using the execution log. RouLette's reinforcement learning-based adaptation approximates the policy that minimizes response time. In this section, we present (i) the requirements for accurately estimating the runtime of global plans thus approximating optimality, and (ii) a reinforcement learning formulation that satisfies the requirements.
Cost estimation: The eddy optimizes the global plan's response time. However, it can observe only intermediate cardinalities in each episode's plans. It estimates time from cardinalities using a cost model. We refer to the estimate as cost. The cost model computes operator a's cost as a function of input and output sizes, $c_a(n_{in}, n_{out})$. The total cost is the sum of all operator costs in a plan.
Requirements: Policies minimize the total cost, which includes the cost of operators later in the plan. Hence, they need to estimate the

long-term effects of decisions, which are caused by the cascading effect of operator selectivity across the plan. For example, in Figure 6's join-phase, the input size for probing $STeM_T$ is 6, whereas, if $STeM_T$ had been probed before $STeM_S$, the input size would have been 5. With 20% larger input, the probe's cost is likely to be higher.

Another long-term effect of decisions is on data distribution due to attribute and join-crossing correlations. Data distribution affects operator selectivity. Assume that in Figure 6's example, only the first 60% of R's vector has matches in $STeM_S$, and only the last 40% has matches in $STeM_T$. Join selectivity is 120% for $R \bowtie S$, 60% for $R \bowtie T$, and 0% for both $(R \bowtie S) \bowtie T$ and $(R \bowtie T) \bowtie T$. Selectivity depends on the predicates of all queries for the input sub-expression, which the virtual vector summarizes.

Also, the eddy optimizes tree-shaped global plans and hence policies affect costs across multiple branches. Long-term cost estimation counts shared operators once for their whole query-set and aggregates cascading costs across all branches e.g. long-term costs for probing S in Figure 6 include the cost of probing S, U, V, T and W, and the cost of routing selections. Then, the policy can choose candidates that minimize the global cost by exploiting sharing, even when they are sub-optimal for individual queries.

Thus, to accurately estimate the best candidate, the policy predicts cascading cardinalities and correlations across all branches. Existing selectivity-based approaches fail the requirements and, as experiments show in Section 6.2, produce sub-optimal and expensive plans. RouLette's policy satisfies all three requirements, by using reinforcement learning on the following MDP.

Formulation: We model multi-step optimization as an MDP. The eddy is an agent that composes plans by choosing one of the candidates at each step. In the following paragraphs, we define the four components of an MDP that optimizes the global plan.

States: States contain the information required to model multi-step optimization. Decisions process states to choose the best candidate.

To express actions, transitions and rewards, the MDP requires the virtual vector and the input size of the current recursive step. The virtual vector determines candidates and the recursive steps that follow. The input size determines the output size given the chosen operator's selectivity, which the virtual vector also affects, and both determine cost estimation when computing rewards. The input size and the virtual vector form an extended vector $(n, \mathcal{L}, Q)$. Later, we show that the specialization can omit input size.

To express recursion, the MDP models all pending recursive steps as a stack of extended vectors, with the current step at the top. The state is the stack and the state space is the set of stacks with elements from $\mathbb{R} \times \mathcal{L}^* \times 2^{Q}$. Our notation represents a stack as $top : tail$ and an empty stack as ϵ. In Figure 7, the state is $(5, \{R\}, \{Q1, Q2, Q3\}) : \epsilon$ for step 1 and $(0, \{R, S, T\}, \{Q1\}) : (5, \{R, S\}, \{Q2, Q3\}) : \epsilon$ for step 3.

Actions: An action chooses a candidate for the current vector i.e. the top of the state's stack. Hence, a state's actions are:

$$A((n, \mathcal{L}, Q) : s_{tail}) = A((n, \mathcal{L}, Q) : \epsilon) = cand(\mathcal{L}, Q)$$

Transitions: Choosing a candidate invokes one or two recursive steps, changing the state. Transitions replace the top of the stack with vectors for the new recursive step(s). For example, $(5, \{R\}, \{Q1, Q2, Q3\}) : \epsilon$ transitions to $(0, \{R, S, T\}, \{Q1\}) : (5, \{R, S\}, \{Q2, Q3\}) : \epsilon$. Operators affect the sizes of new vectors. To express output sizes, we use conditional selectivity $p_{\mathcal{L},Q}(o)$, which models the output to input ratio for operator o and sub-expression results with virtual vector $(\mathcal{L}, Q)$. Candidate o's output is $p_{\mathcal{L},Q}(o) * n$, whereas the routing selection's is $p_{\mathcal{L},Q}(\sigma_{Q-Q_o}) * n$, if any. Sharing pushes one new vector and transitions from $(n, \mathcal{L}, Q) : s_{tail}$ to:

$$(p_{\mathcal{L},Q}(o) * n, \{o\} \cup \mathcal{L}, Q) : s_{tail}$$

Divergence pushes two vectors and transitions to:

$$(p_{\mathcal{L},Q}(o) * n, \{o\} \cup \mathcal{L}, Q \cap Q_o) : ((p_{\mathcal{L},Q}(\sigma_{Q-Q_o}) * n, \mathcal{L}, Q - Q_o) : s_{tail})$$

If there are no more candidates, a *null* action pops the top of the stack, and the state transitions to s_{tail}.

Rewards: An action's reward represents the operator's cost. As reinforcement learning maximizes rewards, operators incur negative rewards. Using the cost model, the reward when sharing is:

$$R((n, \mathcal{L}, Q) : s_{tail}, o) = -c_o(n, p_{\mathcal{L},Q}(o) * n)$$

Divergence also includes the selection's cost, hence the reward is:

$$R((n, \mathcal{L}, Q) : s_{tail}, o) = -c_o(n, p_{\mathcal{L},Q}(o) * n) - c_{\sigma_{Q-Q_o}}(n, p_{\mathcal{L},Q}(\sigma_{Q-Q_o}) * n)$$

4.3 Specialized Q-learning Implementation

The formulation satisfies requirements for modelling the cost of global plans but is difficult to use in practice. The state space is large due to the input-size parameter and the stack representation. In this section, we present the design and implementation of a specialized Q-learning that, by exploiting two properties of cumulative rewards, independence and proportionality, reduces the state space.

Independence: Vectors in the stack have disjoint query-sets hence incur future costs independently across different branches of the plan. The cumulative cost of the state is the sum of cumulative costs for each vector in the stack. To minimize cumulative cost, the eddy separately minimizes the cost of each vector e.g. in step 3, to optimize $(0, \{R, S, T\}, \{Q1\}) : (5, \{R, S\}, \{Q2, Q3\}) : \epsilon$, it optimizes $(0, \{R, S, T\}, \{Q1\}) : \epsilon$ and $(5, \{R, S\}, \{Q2, Q3\}) : \epsilon$. Also, each vector's future costs include only cumulative costs of vectors created by that step's decision. We rewrite decisions and update rules to use only popped and pushed vectors, thus hiding the stack's tail.

Proportionality: Intuitively, operator cost is linear to input size i.e. doubling the input size will roughly double the required computations. Hence, we define the cost model as a linear function:

$$c_a(n_{in}, n_{out}) = \kappa_a * n_{in} + \lambda_a * n_{out}$$

By definition $n_{out} = p_{\mathcal{L},Q}(op) * n_{in}$, so the output size, the cost and hence a vector's cumulative cost is linear to input size. Then, all decisions and updates can be reduced to singleton states $(1, \mathcal{L}, Q) : \epsilon$. Normalizing the Q-values of candidates by input size results in the same decisions e.g. the optimal decision for $(5, \{R\}, \{Q1, Q2, Q3\}) : \epsilon$is the same as for $(1, \{R\}, \{Q1, Q2, Q3\}) : \epsilon$. Also, to express future costs, updates scale Q-values by operator selectivity e.g. for probing $STeM_S$, the future cost is $1.2 * Q((1, \{R, S\}, \{Q1, Q2, Q3\}) : \epsilon)$.

By exploiting independence and proportionality, Q-learning interacts only with $(1, \mathcal{L}, Q) : \epsilon$ states, or simply $(\mathcal{L}, Q)$. We discuss the implementation and integration of the algorithm in RouLette.

Q-table: Q-learning learns $Q((\mathcal{L}, Q), o)$, which is the best-case cumulative cost at $(\mathcal{L}, Q)$ if the policy decides o. The algorithm needs a method for inferring and updating $Q((\mathcal{L}, Q), o)$. We use traditional map-based Q-learning. Deep learning is unsuitable for adaptive processing, as training and inference is prohibitively expensive.

Algorithm 2 Policy implementation

```
1:  procedure NEXT_OPERATOR($\mathcal{L}, Q$)
2:      $cand = cand(\mathcal{L}, Q)$
3:      if $choose - random() == true$ then return $random(cand)$
4:      $return\ argmax_{a \in cand}\{Q(\mathcal{L}, Q, a)\}$
5:  procedure UPDATE($\mathcal{L}, Q, o, n_{in}, n_{out}, n_{div}$)
6:      $r = 0$
7:      $q = max\{Q(\mathcal{L} \cup \{o\}, Q \cap Q_o, a) \mid a \in cand(\mathcal{L} \cup \{o\}, Q \cap Q_o)\}$
8:      $r = r + (-\kappa_o * n_{in} - \lambda_o * n_{out} + \gamma * n_{out} * q)/n_{in}$
9:      if $n_{div} \neq null$ then
10:         $q = max\{Q(\mathcal{L}, Q - Q_o, a) \mid a \in cand(\mathcal{L}, Q - Q_o)\}$
11:         $r = r + (-\kappa_\sigma * n_{in} - \lambda_\sigma * n_{div} + \gamma * n_{div} * q)/n_{in}$
12:     $Q(\mathcal{L}, Q, o) = (1 - \mu) * Q(\mathcal{L}, Q, o) + \mu * r$
```

Map-based Q-learning stores the current $Q((\mathcal{L}, Q), o)$ estimates in a hash map indexed by $(\mathcal{L}, Q), o)$ triplets. As both $\mathcal{L}$ and Q are sets with small domains, we store them as bitsets. Then, concatenating the bytes of $\mathcal{L}$, Q and o forms a unique key for each state. Decisions and update rules use the unique triplets to access the map.

To encourage exploration in early episodes and exploitation in later episodes, we use optimistic initialization [36]. As rewards and Q-table values are negative, we initialize values to zero. The triplet space is partially explored hence Q-table is sparse. We set the map to store only non-zero values and return 0 for failed lookups.

Decisions: Decisions choose one of the candidates. Algorithm 2's *NEXT_OPERATOR* presents decision-making. As $-Q((\mathcal{L}, Q), o)$ is expected cumulative cost, deterministic decisions choose the candidate with the maximum Q-value (line 4). This requires one Q-table access per candidate. Sporadically, with probability ϵ, decisions choose at random to guarantee eventual convergence (line 3).

Updates: By monitoring execution, the eddy generates a log entry for each processed operator o in the following format:

$$(\mathcal{L}, Q, o, n_{in}, n_{out}, n_{div})$$

n_{in}, n_{out}, n_{div} stand for the size of input, o's output and σ_{Q-Q_o}'s output , if any (otherwise *null*). By invoking the update rule for each entry, the eddy adapts the policy.

Algorithm 2's *UPDATE* presents the update rule. The update rule propagates cumulative costs from operators that were added by recursion at $(\mathcal{L}, Q)$. Due to independence, it estimates cumulative rewards for each branch separately (lines 7-8 for $Q \cap Q_o$, lines 10-11 for $Q - Q_o$). Estimation for $Q \cap Q_o$ works as follows: Line 7 estimates, by comparing all $Q((\mathcal{L} \cup \{o\}, Q \cap Q_o), a)$, $a \in cand(\mathcal{L} \cup \{o\}, Q \cap Q_o)$, the best cumulative cost q that recursion can create at $(\mathcal{L} \cup \{o\}, Q \cap Q_o)$. Line 8 adds the cost of o to the estimate in three steps: (i) it multiplies q by n_{out} to undo normalization. (ii) it adds the direct costs of o, and (iii) it normalizes the estimate again by n_{in}. The same estimation method is applied for $Q - Q_o$. r aggregates the total estimate. Thus, Q-learning bootstraps from the current Q-value estimates for state $(\mathcal{L} \cup \{o\}, Q \cap Q_o)$ and, if required, state $(\mathcal{L}, Q - Q_o)$. In the end, the Q-table value is updated to a weighted average of its previous value and the total estimate. After several episodes, Q-learning learns $Q((\mathcal{L}, Q), o)$.

Tuning: Q-learning depends on three hyper-parameters that represents different trade-offs: lowering μ trades off learning speed for smoothing noise due to local data distribution, lowering ϵ trades off exploration for Q-table exploitation, and lowering γ reduces the relative weight of future rewards. As future rewards are equally important, we set $\gamma = 1$. We tune μ and ϵ by using grid search.

We also tune the cost model to emulate execution time. We assume that all operators of the same type, e.g. all joins, have the same κ and λ. To tune the parameters, for each operator type, we measure execution time in nanoseconds for various input and output sizes and apply linear regression to estimate κ and λ. We get: (i) for selections $\kappa = 9.32$ and $\lambda = 4.62$, (ii) for routing selections $\kappa = 3.60$ and $\lambda = 0.92$, and (iii) for joins $\kappa = 38.57$ and $\lambda = 43.29$.

5 ADAPTIVE MULTI-QUERY EXECUTOR

On top of learned policies, RouLette owes its high throughput to efficient shared operators and low-overhead adaptation. In this section, we describe (i) the implementation of shared operators, and (ii) optimizations that eliminate adaptive processing's bottlenecks.

5.1 Efficient Shared Operators

RouLette's selection and join-phases comprise shared operators. In this section, we present operator design and follow the running example's selection-phase in Figure 8 and join-phase in Figure 9.

Selections: Each selection-phase operator filters the query-sets of its input tuples by evaluating one or more predicates. For each input tuple, it computes a predicate result bitset, shown below Figure 8's operators – a set i^{th} bit means that Q_i's predicates in the selection are satisfied. Filtering removes queries with zero bits from the tuple's query-set by computing the bitwise AND of the bitsets. The new bitset, which stands for query-set intersection, is the output's query-set. Selection drops tuples with empty query-sets.

To reduce shared selection costs, RouLette batches predicate evaluation on each attribute using grouped filters e.g. $\sigma_{R.d}$ evaluates $Q1$'s and $Q3$'s predicates (and *true* for $Q2$) at once. Prior work [24] prunes comparisons by indexing predicates using structures such as search trees. In that case comparisons are still linear to satisfied queries and hence to all queries. RouLette uses a novel evaluation method, whose cost is logarithmic to query count, for arithmetic or dictionary comparisons. It constructs lookup tables, depicted above operators in Figure 8, that store precomputed predicate results for ranges or values. Predicate evaluation requires a binary search.

STeMs: RouLette uses a shared STeM for each relation across all queries and joins. The STeM stores selection-phase result tuples and, on each join key, builds indices for joins e.g. hash-index for equi-joins. To reduce footprint, we use unified STeM entries:

(index-vector, vID, timestamp, query-set)

STeMs stores entries as a contiguous memory block. The inserted tuple consists of *vID* and *query-set*. Each of STeM's index uses one element of *index-vector* to build a self-referential data structure e.g. list-based hash bucket for hash-indices. The *index-vector* also stores the join key to avoid late materialization for STeM tuples' attributes. Finally, STeM uses *timestamp* to ensure insert-probe atomicity. Figure 9 shows STeMs above each probe. In our example, S has equi-joins on $S.a$, $S.e$ and $S.f$ and builds hash-indices.

Probes search the STeM for matching tuples, inserted in previous episodes, and produce concatenated probe-match pairs. Vector 2 in Figure 9 is the result of probing $STeM_S$ for R's vector. Probes

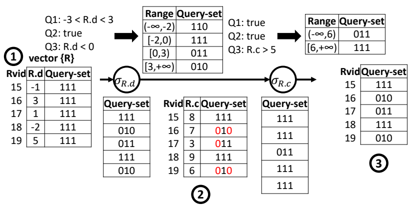


Figure 8: Execution of selection-phase

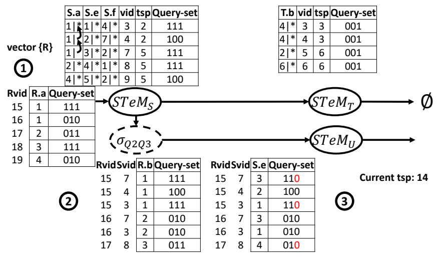


Figure 9: Execution of join-phase

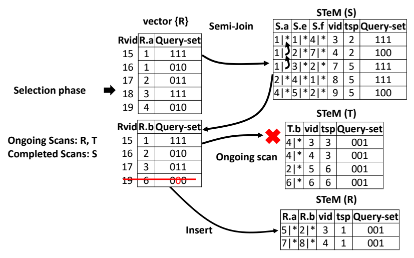


Figure 10: Symmetric Join Pruning

use STeM indices to efficiently find matches. Then, they compare timestamps to enforce insert-probe atomicity – only matches with older timestamps are considered. Finally, they compute query-sets of probe-match pairs by intersecting the query-sets of the probing and the probed tuples, i.e., bitwise AND of the bitsets, and discard pairs with empty query-sets, such as R's tuple 19 with S's tuple 9.

Routing selections: Selections in the join-phase permit tuples of specified queries to pass, e.g. in Figure 9, it retains tuples from $Q2$ and $Q3$. A bitwise AND with a filter mask clears other queries from the bitset. Such selections reduce downstream processing.

Router: Routers send shared output to the host, by multicasting tuples to their query-set's RouLette sources. To increase output locality and reduce cache and TLB misses, they adapt the design of two-pass partitioning to multicasting. Hence, routers increase cache hits.

5.2 Optimizations for Adaptive Processing

Adaptive processing suffers from overhead due to STeM materialization and versioning, and lack of projectivity. To match optimize-then-execute performance, RouLette uses novel optimizations.

Symmetric Join Pruning: Symmetric joins require that all relations be materialized and hence incur materialization overhead. To reduce the overhead, RouLette materializes only tuples that can form output tuples for their query-set. We call this *symmetric join pruning*. Figure 10 shows pruning for the symmetric join of Figure 6. In the example's episode, the symmetric join processes R's vector. Tuple 18 has no matching entry in S. As all queries contain $R \bowtie S$ and the $STeM_S$ is final, pruning infers that 18 cannot form any output tuple and drops 18 before insertion. Also, it infers that 19 cannot form output tuples for Q1 and Q2, hence it adjusts the query-set. As the new query-set is empty, pruning drops 19. Pruning cannot use $STeM_T$, because T's scan is ongoing; future inserts can yield matches for 15-17. To drop tuples and modify query-sets, pruning uses semi-joins with fully-ingested joinable STeMs. RouLette integrates semi-joins into the selection phase as filters.

Pruning emulates filtering in non-left deep plans that use join results as inner relations. STeMs store semi-join results hence probes and semi-joins with pruned STeMs return even fewer matches. Filtering propagates across the plan, beyond direct joins and STeMs store the results of semi-join trees. Still, symmetric joins require extra probes to construct results. Caching intermediate results [5] eliminates extra probes and is complementary to RouLette.

As pruning requires fully-ingested relations, to increase pruning opportunities, RouLette controls the order in which ingestion initiates circular scans. It chooses the order based on three insights: (i) Small relations that are on the build-side in all joins should be ingested first. (ii) Ingesting large relations should be postponed, as they are the targets of pruning. (iii) M:N semi-joins are avoided, as they are expensive. The insights apply to common schemas that use dimension tables (e.g., star, snowflake, snowstorm).

To choose the order, RouLette ranks relations using a heuristic. The heuristic, starting from rank 1, works as follows: (i) it marks unranked relations that are smaller than all other joinable unranked relations. (ii) it assigns the current rank to marked relations and increments the current rank. (iii) it adjusts cardinality estimates, based on pruning, and repeats the steps. Ranking produces a partial order of scans for each scheduled batch. Except for having its left-most relation fixed, join order is orthogonal to scan order.

Scalable versioning: RouLette parallelizes episode execution. Critical sections, such as ingestion and policy updates, are rare, and hence are lock-based. The main point of contention are STeMs.

To reduce contention and scale up, RouLette's STeMs use wait-free indexing and batch versioning. First, wait-free indices use atomics and hence reduce insert/probe contention. Second, batch versioning reduces contention on version counter, as it requires only two atomics per vector. For batch versioning, STeMs use both local and global versions. Inserts use the same STeM-local version for each vector's tuples. Then, they map the STeM-local version, by default globally invalid, to a global version. To check atomicity conditions, each probe translates STeM-local to global timestamps.

Adaptive projections: As adaptive processing lacks projections, probe results grow increasingly wide hence materializing intermediate vectors becomes more expensive. To drop redundant columns and reduce materialization, RouLette introduces adaptive projections. By identifying columns used by downstream operators in the episode's plans, it keeps a minimal set of vIDs and sheds the rest.

6 EVALUATION

We evaluate a prototype of RouLette. The experiments show (i) RouLette's ability to scale throughput, (ii) RouLette's performance gains over online sharing and query-at-a-time DBMS, (iii) the benefit of learned policies over selectivity-based policies, (iv) the impact of timing dependencies to sharing and learning, (v) the sensitivity of learning rate to workload characteristics, (vi) the effect of executor optimizations, and (vii) RouLette's scalability in multi-core CPUs.

The experiments run on a two-socket server with 12-core Intel Xeon E5-2650L v3 CPUs running at 1.8 GHz and 256 GB of DRAM. The server uses Ubuntu 18.04 LTS and GCC 7.4.0. RouLette affinitizes threads and memory to one NUMA node. With the exception of Section 6.4, experiments use one worker. In all experiments, reported numbers are the average of five runs.

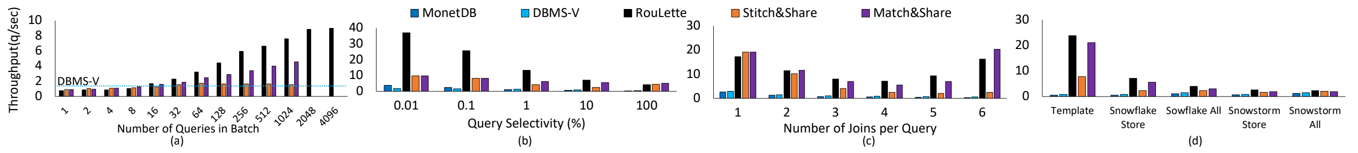


Figure 11: Sensitivity analysis: varying (a) concurrency (b) selectivity (c) number of joins (d) schema type

The experiments use data from TPC-DS [31] (scale factor 10, 8.65 GB in-memory) and Join Order Benchmark, or JOB, [22] (1.79 GB in-memory). To evaluate RouLette for a wide range of workloads, we generate a pool of thousands of queries on TPC-DS schema, with different joins and predicates. To assess learned policies, we use JOB as it uses real data that violates assumptions that oversimplify optimization. JOB comprises 113 SPJ queries with 3-16 joins.

We assess RouLette's ability to optimize any workload it is given. Thus, we use workload-agnostic scheduling. By sampling queries, we produce a query stream on which RouLette uses FIFO batching.

All experiments use the same Q-learning hyper-parameters. To tune μ and γ, we use grid search to minimize the total response time for five batches of 64 JOB queries. We get $\mu = 0.21$ and $\epsilon = 0.014$.

6.1 Throughput Evaluation

Methodology: This section shows RouLette's performance improvement. We compare RouLette against two query-at-a-time systems, a vectorized DBMS (DBMS-V) and MonetDB. We also compare against two online sharing techniques, Stitch& Share and Match& Share. Stitch& Share composes global plans by sharing common sub-trees between individual plans produced by PostgreSQL. It is used in QPipe [16], SharedDB [13]. Match& Share adds each query to the global plan with minimum additional cost. It is used in DataPath [2]. To execute global plans in a common shared engine, we implement a prototype that uses the batched execution model [13] and adopts all useful optimizations and operators from RouLette.

To assess RouLette, the performance comparison unfolds in two steps: (i) a sensitivity analysis spanning large diverse multi-query workloads, (ii) JOB workloads. Shared-work systems execute each workload's queries as a single batch, whereas query-at-a-time systems execute queries one after the other. The compared metric is throughput, i.e. number of queries over total execution time.

We omit a comparison against offline sharing, as it cannot scale to a meaningful batch size. SWO [14], a state-of-the-art algorithm, takes 137 seconds to optimize a batch of 11 queries, with 4 joins each. The 11-query batch is the largest SWO could optimize with an one-hour timeout. For this small batch, RouLette's throughput is only 4% lower than SWO's, whereas, for online approaches, throughput is at least 7% lower. As this Section's experiments show, for larger batches, the gap between RouLette and online sharing widens.

Sensitivity Analysis: The experiment examines RouLette's batch execution performance under varying workload conditions. The varying conditions are the batch size, the selectivity and the number of joins of individual queries, and the schema type. To generate required workloads, we implement a query generator that takes the conditions as parameters.

The query generator uses a two-step process: (1) it chooses a subgraph of the schema as a join graph. It does not join fact tables of different channels [28]; this occurs only in query 78 of TPC-DS. (2) it produces predicates to match a target selectivity. To precisely control selectivity, we extend each TPC-DS table with a uniformly distributed column with values from 0 to 999 and produce *BETWEEN* predicates. The predicates are applied to 3 of the query's relations, chosen randomly, and have unequal selectivity.

Figures 11a-11d show the throughput. In each Figure, three parameters are constant and the fourth varies. The default values are 10% selectivity, 4 joins, Store snowflake sub-schema (similar to Star Schema and TPC-H), and 512 queries. The query generator produces 4096 queries per configuration and forms batches by sampling the queries without replacement.

Varying concurrency: Figure 11a shows that RouLette's throughput scales with increasing batch sizes. The available memory restricts the maximum batch size per system. Shared approaches improve their throughput as a function of the batch size because sharing opportunities are increased. RouLette's throughput grows faster, as it discovers more opportunities, and, despite adaptation overhead, overtakes Stitch& Share and Match& Share after 16 and 32-query batches respectively. RouLette's maximum speedup is 10.70 over DBMS-V, the faster of the two DBMS, whereas online sharing's maximum speedup is 3.65. Thereupon, RouLette's throughput hits a plateau when query-set operations dominate execution time. As the cost of query-set operations grows linearly to batch size, the plateau is a bottleneck of the Data-Query model. To further scale throughput, future work needs to revise the Data-Query model.

Varying selectivity: Figure 11b shows higher throughput for all selectivities. Response time is increased as a function of selectivity, hence throughput is decreased. RouLette exploits more opportunities compared to online sharing approaches; Stitch& Share misses opportunities as predicates produce different plans for the same join set, whereas for low selectivity Match& Share fails to exploit sub-expressions in the existing global plan when planning each query. For queries without filters (100% selectivity), their opportunity detection limitation is lifted. Out of the query-at-a-time DBMS, MonetDB performs better for low selectivity, but suffers from intermediate materializations for higher selectivity, unlike DBMS-V.

Varying number of joins: Figure 11c shows that sharing is sensitive to join-set diversity. The number of distinct join-sets is maximum for 3-4 joins, whereas including few or almost all joins increases homogeneity (all 6-join queries have the same join set). Each system's throughput depends on whether sharing can offset increasing join processing costs. RouLette's throughput reflects the effect of homogeneity, decreasing until 3-4 joins and then increasing. It outperforms online sharing when heterogeneity is high, because it reorders joins to discover opportunities, and benefits as homogeneity is again increased. However, increasing homogeneity benefits Match & Share as well, as it can also reorder joins to a smaller extent. Match & Share retakes the lead for 6-join queries.

Varying the schema: Figure 11d shows that RouLette works best for homogeneous workloads, but is still effective for diverse queries.

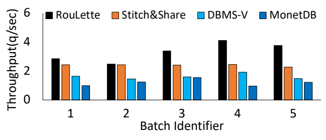


Figure 12: JOB batches

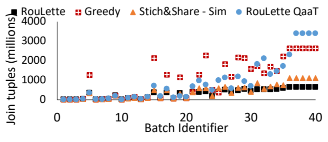


Figure 13: Policy - batches

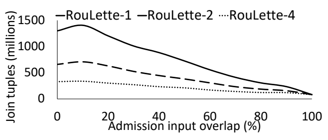


Figure 14: Policy - dynamic

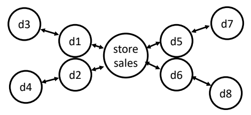


Figure 15: Synthetic schema

The five workloads comprise queries whose join-set is: (i) *store_sales* ⋈ *date_dim* ⋈ *hdemo* ⋈ *item* ⋈ *customer* (*template*), (ii) subgraph of Store snowflake sub-schema (*snowflake-store*), (iii) subgraph of any channel's snowflake sub-schema (*snowflake-all*), (iv) subgraph of Store snowstorm sub-schema (*snowstorm-store*), and (v) subgraph of any channel's snowstorm sub-schema (*snowstorm-all*). RouLette's speedup decreases with high join-set diversity, as queries have little work in common. For *snowstorm-all*, the most diverse case, RouLette overtakes DBMS-V after 32-query batches. However, batch size compensates for diversity. RouLette increases throughput by 3.15x for snowstorm-store and by 1.57x for snowstorm-all, and it outperforms online sharing, despite materializing fact tables.

Large Queries with Correlated Data Figure 12 shows that RouLette improves performance, even for queries with many joins and data correlations that are challenging for optimizers. The workload comprises 64-query batches produced by sampling JOB. RouLette outperforms both DBMS and online sharing. Speedup echoes the results of *snowstorm-all*. The experiment excludes Match& Share, as its custom optimizer supports only uniform data, for which it can estimate the intermediate result overlap between different queries.

Summary RouLette outperforms query-at-a-time DBMS in all cases and online sharing when the optimal plan is non-trivial. In workloads with diverse join sets and schemas, it exposes more opportunities that online sharing misses. Still, diversity reduces opportunities for all sharing techniques; increasing homogeneity using workload-aware batching is a promising optimization. Finally, it achieves scalability, as it increases throughput with increasing query counts until Data-Query model dominates execution.

6.2 Quality of Learned Planning

Methodology: The experiment evaluates the ability of policies to exploit opportunities in both static workloads, processed in batches, and dynamic workloads, with runtime query admissions. It focuses on stand-alone policies hence measures the number of intermediate tuples in joins, which is an implementation-independent metric for cost. To compute the metric, we add up the log's output vector sizes. As joins dominate execution time, we exclude selections.

Static Opportunities: The experiment compares the behavior of different policies. RouLette processes workloads in batches. It schedules all queries at once hence fully sharing all scans and common intermediate tuples. We generate 5 batches for each of sizes 1, 2, 4, 8, 16, 32, 64, and 113 by sampling JOB queries without replacement.

Figure 13 shows the cost for varying batch sizes. Each batch corresponds to a sequence number based on the size. Size 1 maps to the range $[1, 5]$, size 2 to $[6, 10]$, …, size 113 to $[36, 40]$. Batches 36-40 are identical to each other. The figure includes 4 different configurations. *RouLette* is the learned policy. *Greedy* is the selectivity-based policy from CACQ and CJOIN. By choosing plans independently for each query using the learned policy and then sharing common sub-expressions, *Stitch&Share - Sim* simulates Stitch&Share. Finally, *RouLette QaaT* is the cumulative cost of executing queries one after the other. RouLette reduces the cost in all cases compared to RouLette QaaT.

Learning vs Selectivity: The results show that learned policies choose superior plans. The *Greedy* policy incurs comparable cost to the learned policy for small batches, but suffers from high-cost outliers. Outliers include ≈ 7% of single JOB queries. As the batch size is increased, optimization hazards are increasingly likely to occur hence penalize most batches. For 64-query batches, the learned policy produces 3.24*x* fewer intermediate tuples on average.

Learning scope: Results also show that a global learned policy (*RouLette*) outperforms a query-local policy (*Stitch&Share – Sim*), as it considers sharing during planning and is preferable to individual decisions. While the cost is similar up to 8-query batches, *RouLette* produces 1.71*x* fewer tuples for 113 queries.

Dynamic Opportunities: Figure 14 shows the interplay between sharing and learning. RouLette admits instances of JOB query 17a one-at-a-time or in batches. We measure the percentage of overlapping input between back-to-back admissions: 0% is query-at-a-time execution, whereas 100% is single batch execution. The intermediate tuples are decreased as a function of the overlap. For small overlaps, the cost is increased because sharing does not compensate for restarting learning. An overlap of 10% increases cost by 8% for single-query admissions. Batching reduces the cost of processing, as it reduces interference and guarantees opportunities. Admitting four-query batches for every 40% of the input produces 1.4*x* fewer tuples, compared to admitting single queries for every 10%.

Learning Rate: The next experiment evaluates the ability of the policy to learn plans for workloads with varying complexity. To vary complexity, we modify TPC-DS. Queries join *store_sales* with chains of synthetic relations. At each step, the policy considers one candidate per chain. We generate synthetic relations by sampling *date_dim* with replacement at varying rates. To ensure a large difference between each query's best and worst plan, rates are 0.4-0.6 for half of the chains and 1.7-2.5 for the other half. Each query spans half of the join graph and includes an equal number of high and low-selectivity joins. We generate workloads with varying number of chains and relations. Figure 15 shows the schema for workload "Chains=4,Relations=9".

Figures 16a-16h show the policy's convergence across the episode sequence for 64-query batches from various workloads. Plots include each workload's parameters. For each episode, it plots the measured cost and the policy's estimate of the minimum cost. As execution progresses and future costs are propagated, the policy's estimate is increased and measured cost is decreased; when they converge, the policy is optimal. The experiment shows that convergence is slower when the state space is broader (more candidates) and deeper (join size). Figures 16a-16c show that, when candidates are few, the policy converges fast even for large joins. By contrast, Figures 16d-16h show that, when candidates are many, the policy

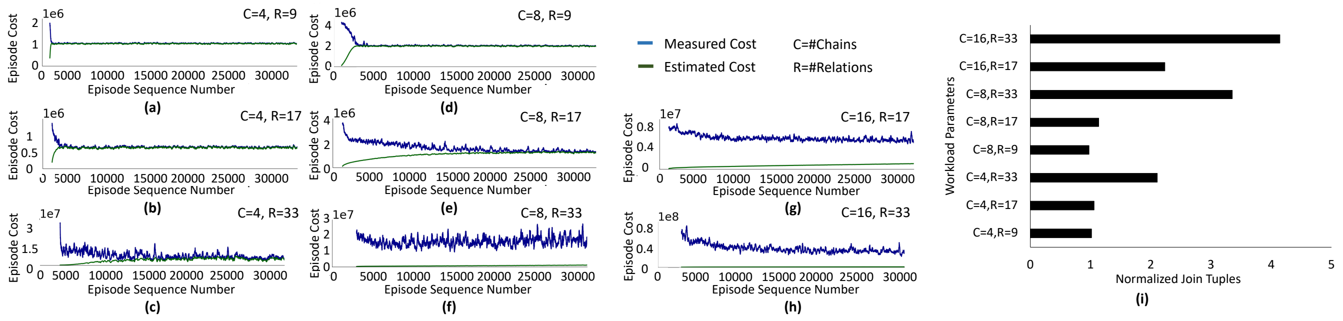


Figure 16: (a)-(h) Episode cost comparison: measured vs estimated, (i) Learned vs Selectivity-based policy

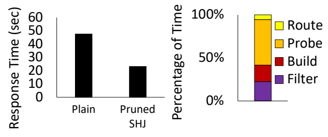


Figure 17: JOB batch profile

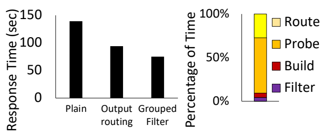


Figure 18: Large batch profile

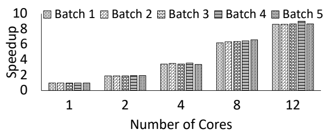


Figure 19: Varying cores

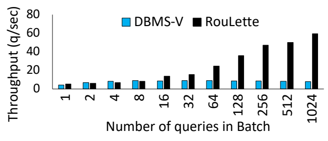


Figure 20: Interference

converges only for small joins. Figure 16i plots the intermediate join tuple ratio for RouLette over the Greedy policy. As joins have no data correlations, Greedy is near-optimal. The comparison shows that, when convergence is slow, learned policies suffer from exploratory decisions. To mitigate the effects of slow convergence and to learn plans for large schemas, we aim to extend the policy with mechanisms, such as heuristics.

Summary: Sharing-aware learned policies substantially improve adaptive processing. They produce fewer tuples compared to selectivity-based policies and to sharing-oblivious learned policies. Learned policies permit query admissions at runtime but suffer from interference when the overlap is low. Batching admissions reduces interference and increase sharing. Finally, learned policies typically converge within few thousands episodes, but suffer from slow convergence for workloads on large schemas.

6.3 Effect of Optimizations

Figures 17 and 18 show the benefit from individual optimizations, applied incrementally, and the breakdown of the execution time after applying the optimizations. The experiment analyzes two batches, a 64-query batch of JOB queries and a 512-query batch of generated queries (default parameters). Joins dominate execution for both batches. For the JOB batch, pruning is the most important optimization, giving 2.05 speedup. For the synthetic batch, RouLette's novel router and grouped filter algorithms are the most important optimizations: together they result in 1.85 speedup.

6.4 Multi-core Execution

In this section, we evaluate RouLete's performance when using multi-core CPUs. RouLette scales up in one NUMA socket.

Figure 19 shows RouLette's speedup as the number of workers is increased from 1 to 12. The experiment uses the five batches of 64 JOB queries from Figure 12. Speedup is increased monotonically for all batches and reaches 8.63-9.04 (71.9-75.3% efficiency).

Figure 20 shows that, for concurrent execution, DBMS-V's throughput suffers due to inter-query interference, whereas RouLette's benefits. DBMS-V receives and processes queries from 1 to 1024 clients. When using one client, DBMS-V uses data-parallelism. For more clients it shares resources across queries. Clients run isolated in the remote NUMA node. Concurrent execution initially improves throughput up to 2.06x. However, after 64 clients, throughput is gradually decreased due to interference. DBMS-V runs out of memory after 1024 queries. RouLette uses all cores for processing query batches; each batch contains one query per client. RouLette's speedup over DBMS-V is increased as a function of concurrency.

7 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their thoughtful comments. Also, we would like to thank Georgios Giannikis and Gustavo Alonso for providing access to SWO [14]. Finally, we would like to thank our friends and colleagues Stella Giannakopoulou, Matthaios Olma, Ioannis Mytilinis, Viktor Sanca, Srinivas Karthik, Bikash Chandra, Periklis Chrysogelos, Konstantinos Koukas, Aggelos Biboudis, Panayiotis Smeros, and Christos Vrachas for their valuable feedback. This work is funded by SNSF project "Efficient Real-time Analytics on General-Purpose GPUs", subside no 200021_178894/1 and the 2020 Facebook Fellowship Program.

8 CONCLUSION

We have presented RouLette, an adaptive multi-query multi-way join operator that tackles the limitations of online and offline sharing. Rather than follow an optimize-then-execute approach, RouLette uses runtime adaptation to move sharing-aware optimization out of the critical path, restoring scalability. It progressively explores sharing opportunities using a heuristic based on reinforcement learning. RouLette also proposes optimizations that reduce the adaptation overhead. The experiments show that RouLette scales to hundreds of complex queries, unlike offline sharing, and improves throughput compared to query-at-a-time and online sharing systems. Hence, it makes inroads on the long-standing problem of building scalable high-throughput analytical systems.

REFERENCES

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 169–180.

[2] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Perez. 2010. The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/1807167.1807224

[3] Ron Avnur and Joseph Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. *ACM SIGMOD* 29. https://doi.org/10.1145/342009.335420

[4] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive Ordering of Pipelined Stream Filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. ACM, New York, NY, USA, 407–418. https://doi.org/10.1145/1007568.1007615

[5] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. 2005. Adaptive Caching for Continuous Queries. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, USA, 118–129. https://doi.org/10.1109/ICDE.2005.15

[6] Pedro Bizarro and David Dewitt. 2006. Adaptive and robust query processing with SHARP. (05 2006).

[7] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 277–288. https://doi.org/10.14778/1687627.1687659

[8] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *PVLDB* 12(12) (2019), 2022–2034. https://dl.acm.org/citation.cfm?id=3360438

[9] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. 2001. Pipelining in Multi-query Optimization. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Santa Barbara, California, USA) *(PODS '01)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/375551.375561

[10] Amol Deshpande and Joseph M. Hellerstein. 2004. Lifting the Burden of History from Adaptive Query Processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 948–959.

[11] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive Query Processing. *Found. Trends databases* 1 (01 2007), 1–140. https://doi.org/10.1561/1900000001

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

[13] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries with One Stone. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 526–537. https://doi.org/10.14778/2168651.2168654

[14] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared Workload Optimization. *Proc. VLDB Endow.* 7, 6 (Feb. 2014), 429–440. https://doi.org/10.14778/2732279.2732280

[15] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. *SIGMOD Rec.* 28, 2 (June 1999), 287–298. https://doi.org/10.1145/304181.304208

[16] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. New York, NY, USA, 383–394.

[17] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *Proc. VLDB Endow.* 11, 7 (March 2018), 800–812. https://doi.org/10.14778/3192965.3192971

[18] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[19] Sailesh Krishnamurthy, Michael Franklin, Joseph Hellerstein, and Garrett Jacobson. 2004. The Case for Precision Sharing. (10 2004). https://doi.org/10.1016/B978-012088469-8.50085-1

[20] Sanjay Krishnan, Zongheng Yang, Kenneth Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning.

[21] Ramon Lawrence. 2008. Using slice join for efficient evaluation of multi-way joins. *Data Knowl. Eng.* 67 (10 2008), 118–139. https://doi.org/10.1016/j.datak.2008.06.001

[22] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[23] Q. Li, M. Shao, V. Markl, K. Beyer, L. Colby, and G. Lohman. 2007. Adaptively Reordering Joins during Query Execution. In *2007 IEEE 23rd International Conference on Data Engineering*. 26–35.

[24] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/564691.564698

[25] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2016. MQJoin: Efficient Shared Execution of Main-memory Joins. *Proc. VLDB Endow.* 9, 6 (Jan. 2016), 480–491. https://doi.org/10.14778/2904121.2904124

[26] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[27] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Houston, TX, USA) *(aiDM'18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3211954.3211957

[28] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1049–1058.

[29] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. 2020. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. 154–157.

[30] J. Park and A. Segev. 1988. Using common subexpressions to optimize multiple queries. In *Proceedings. Fourth International Conference on Data Engineering*. 311–319.

[31] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, Taking Decision Support Benchmarking to the next Level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 582–587. https://doi.org/10.1145/564691.564759

[32] Vijayshankar Raman, Amol Deshpande, and J.M. Hellerstein. 2003. Using state modules for adaptive query processing. 353– 364. https://doi.org/10.1109/ICDE.2003.1260805

[33] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/342009.335419

[34] Timos K. Sellis. 1988. Multiple-query Optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. https://doi.org/10.1145/42201.42203

[35] Kyuseok Shim, Timos Sellis, and Dana Nau. 1994. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data Knowl. Eng.* 12, 2 (March 1994), 197–222. https://doi.org/10.1016/0169-023X(94)90014-0

[36] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press.

[37] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. 2003. Maximizing the Output Rate of Multi-way Join Queries over Streaming Information Sources. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 285–296. http://dl.acm.org/citation.cfm?id=1315451.1315477

[38] Carl Waldspurger and William Weihl. 2001. Lottery Scheduling: Flexible Proportional-Share Resource Management. (11 2001).

[39] Christopher Watkins. 1989. Learning From Delayed Rewards. (01 1989).

[40] A.N. Wilschut and Peter Apers. 1992. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases - DPD*, 68–77. https://doi.org/10.1109/PDIS.1991.183069

[41] Z. Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13 (2019), 279–292.

[42] X. Yu, G. Li, C. Chai, and N. Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1297–1308.

[43] Jingren Zhou, Per-Ake Larson, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient Exploitation of Similar Subexpressions for Query Processing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. ACM, New York, NY, USA, 533–544. https://doi.org/10.1145/1247480.1247540

[44] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 723–734.