WPI-CS-TR-12-07 November 2012

Robust Distributed Stream Processing

by

Chuan Lei
Elke A. Rundensteiner
and Joshua D. Guttman

Robust Distributed Stream Processing

Chuan Lei, Elke A. Rundensteiner, and Joshua D. Guttman
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609
Tel.: (508) 831–5857, Fax: (508) 831–5776
{chuanlei, rundenst, guttman}@cs.wpi.edu

Abstract

Distributed stream processing systems must function efficiently for data streams that fluctuate in their arrival rates and data distributions. Yet repeated and prohibitively expensive load re-allocation across machines may make these systems ineffective, potentially resulting in data loss or even system failure. To overcome this problem, we instead propose a load distribution (RLD) strategy that is robust to data fluctuations. RLD provides ϵ-optimal query performance under load fluctuations without suffering from the performance penalty caused by load migration. RLD is based on three key strategies. First, we model robust distributed stream processing as a parametric query optimization problem. The notions of robust logical and robust physical plans then are overlays of this parameter space. Second, our Early-terminated Robust Partitioning (ERP) finds a set of robust logical plans, covering the parameter space, while minimizing the number of prohibitively expensive optimizer calls with a probabilistic bound on the space coverage. Third, our OptPrune algorithm maps the space-covering logical solution to a single robust physical plan tolerant to deviations in data statistics that maximizes the parameter space coverage at runtime. Our experimental study using stock market and sensor networks streams demonstrates that our RLD methodology consistently outperforms state-of-the-art solutions in terms of efficiency and effectiveness in highly fluctuating data stream environments.

1 Introduction

Motivation. Distributed stream processing systems (DSPSs) are designed to execute continuous queries over streams of tuples [1, 2, 3, 4]. Continuous queries place heavy workloads on precious system resources from CPU processing cycles, memory, and network bandwidth. Since workloads can vary in unpredictable ways, exploiting the limited resources requires robust and effective load distribution techniques. Figure 1 depicts a typical distributed stream processing system.

Load distribution, the placement of the operators in a query plan to machines (nodes) in the distributed system, is an important design decision, impacting the query processing performance [3]. A carefully selected operator placement may later produce poor performance due to time-varying, unpredictable stream fluctuations. Data fluctuations may in fact necessitate repeated expensive load redistributions in such distributed systems. Such load redistribution may cause delay and potentially make the system fail to react to short-term load fluctuations in time. Example 1 illustrates this problem using a real-world application.

Example 1. Consider a query monitoring stocks that exhibit "bullish" patterns (upward price movement in the stock market) [5] in recent business news and research.

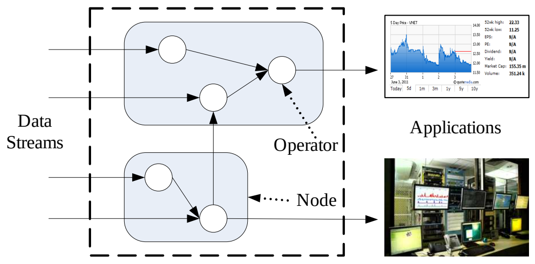
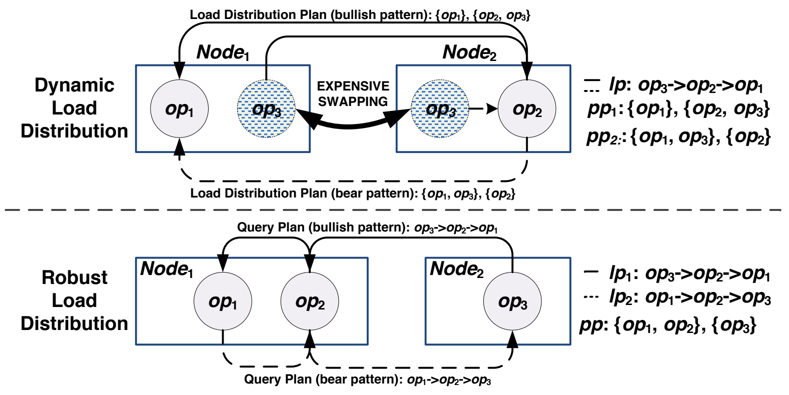


Figure 1. Distributed Stream Processing System

```
SELECT S.company_name, S.symbol, S.price
FROM Stock as S, News as N, Research as R
WHERE matches(S.data, BullishPatterns)          /*op1*/
AND contains(S.sector, News[1 hour])            /*op2*/
AND contains(S.company_name, News[1 hour])      /*op3*/
WINDOW 60 seconds
```

The lookup table *BullishPatterns* contains "bullish" patterns of stock behavior, e.g., "symmetrical triangle". Operator op_1 performs a similarity-based join on the latest stock data tuples from the last 60 seconds with the *Patterns* table. Operators op_2 and op_3 perform matches on the stock sector and the company name with news and research streams. Let c_i and δ_i denote the processing costs and current selectivity of op_i respectively.



Load Distribution Plan (bullish pattern): {op1}, {op2, op3}
Dynamic Load Distribution
Node1
Node2
op1
op3
EXPENSIVE SWAPPING
op3
op2
lp: op3->op2->op1
pp1: {op1}, {op2, op3}
pp2: {op1, op3}, {op2}
Load Distribution Plan (bear pattern): {op1, op3}, {op2}
Query Plan (bullish pattern): op3->op2->op1
Robust Load Distribution
Node1
op1
op2
Node2
op3
lp1: op3->op2->op1
lp2: op1->op2->op3
pp: {op1, op2}, {op3}
Query Plan (bear pattern): op1->op2->op3

Figure 2. Dynamic vs. Robust Load Distribution

Suppose it is a bullish market (i.e., stocks are doing well) and the following condition holds: $\delta_1 > \delta_2 > \delta_3$. Given these statistics, the best query processing order (query plan) is op_3, op_2, op_1 ($c_1 > c_2 > c_3$). Assume we have two machines, n_1 and n_2, with resources r_1 and r_2 (i.e. CPU, memory and network bandwidth) available for processing. Then the best load distribution plan is op_1 on node n_1, and op_2 and op_3 on node n_2, as depicted in the upper illustration of Figure 2. Now suppose breaking news report poor stock performance. This will likely result in fewer matches with the *Patterns* table and instead more matches with news and blogs. In this case, δ_1 will be relatively lower than δ_2 and δ_3 for such data tuples. So plan op_3, op_2, op_1 is no longer the most efficient ordering. In fact, op_2 and op_3 may overload n_2, namely $c_2 + c_3 > r_2$. Consequently, the DSPS has to relocate op_3 to n_1. However, in the future if the market exhibits bullish pattern again, the optimizer might need to revert back to the original query processing order. Then the query executor would again need to move op_3 back to n_2. In this particular scenario, we notice that the load distribution plan, op_1 and op_2 on n_1, and op_3 on n_2

(lower Figure 2) would have been **robust** to support both query processing orders op_3, op_2, op_1 and op_1, op_2, op_3 in different scenarios. Proactive design of this load distribution plan achieves two objectives, i.e., robust query processing performance and tolerance to data fluctuations without dynamic load redistribution.

Insufficiency of State-of-The-Art. Traditional distributed and parallel systems [6, 7] categorize load distribution solutions as either dynamic or static. Dynamic load distribution [1, 2] instead repeatedly improves the current load distribution plan by moving operators across machines to adapt to load changes. However, it comes with several drawbacks, including (1) amortized overhead of repeated load redistributions, (2) adaptation delays, when redistribution opportunities could be missed, and (3) expensive maintenance of statistics.

Traditional static optimizers [8] determine a $single$ "best" (i.e., cheapest overall execution costs) placement of query operators at compile time based on the average estimated statistics of data streams. While this approach imposes low optimization overhead, it cannot adapt the load distribution to changing statistics of data streams such as variations in input rates and selectivities. The most recent work, resilient operator distribution (ROD) [9], aims to produce a feasible physical plan to be resilient to time-varying and unpredictable workloads. However, shortcomings of ROD include: (1) it only focuses on the physical operator distribution without taking logical query plan ordering problem into consideration, (2) its operator distribution plan does not guarantee the given query processing performance, and (3) it is not proactive to changing workloads. A detailed comparison can be found in Section 7.

Our Proposed RLD Approach. Given these inevitable disadvantages of existing load distribution methods, we now propose an end-to-end solution called *Robust Load Distribution* (RLD) that exploits the key principles from load distribution methods while overcoming their respective shortcomings by integrating parametric query optimization. As foundation of RLD, we introduce the multi-dimensional parameter space model, a representation of the uncertainties in statistical estimates of streaming data. We then introduce the dual model of matching *robust logical* and *robust physical* plans which together gracefully handle the fluctuations experienced by the multi-dimensional parameter space. RLD serves as a practical compromise between the two extremes of either static or dynamic optimization.

Considering the intractable complexity of the search space composed of all combinations of logical plans with associated physical plan, we adopt a two-step query optimization approach. First, our proposed algorithm, Early-terminated Robust Partition (ERP), identifies a set of robust logical plans that together cover the parameter space with a *probabilistic guarantee* on the percentage coverage of the parameter space. In particular, each robust logical plan covering a non-trivial area of the space will be found with high probability. A combination of robust logical plans that together cover the space and guarantee the robust performance under any known fluctuations is called a *robust logical solution*.

Next we propose a load distribution algorithm, $OptPrune$ that efficiently produces an optimal *robust physical plan* (i.e., operator allocation plan) that supports the logical plans in a robust logical solution based on their probability of occurrence at runtime and their respective area of robustness in the parameter space. This results in a single robust physical plan tolerant to expected data statistic deviations at runtime.

Contributions. We introduce an end-to-end solution that overlays robust logical plans with a single physical plans in a parameter space. The contributions of this work can be summarized as follows:

1. We introduce the property of robust logical query plans (i.e., ϵ-robustness). We efficiently compute a multi-dimensional parameter space representing uncertainties in statistic estimates, including stream input rates and query operator selectivities.

2. ERP, our robust logical solution algorithm efficiently finds multiple robust logical plans that together assure coverage across the entire parameter space, unattainable by a single plan. ERP forms a *probabilistic bound* on the total uncovered area. Individual robust plans with any non-trivial area will be found with high probability.

3. Given a robust logical solution, we design a family of algorithms that cover the spectrum from the optimization complexity to result optimality. $GreedyPhy$ finds a robust physical plan in polynomial time, whereas $OptPrune$ is a branch-and-bound algorithm using $GreedyPhy$ as bound, which succeeds to significantly bound

the search without compromising optimality.

4. Our performance evaluation using streaming data from stock market and sensor networks confirms that our RLD approach is superior to state-of-the-art solutions [2, 9] in all aspects, including average tuple processing time, total system throughput, and robustness to data fluctuations.

The rest of this paper is organized as follows. We define the RLD problem in Section 2, and overview our solution in Section 3. Sections 4 and 5 describe our algorithms for generating a robust logical solution and the associated robust physical solution. The experimental results are presented in Section 6. Sections 7 and 8 discuss related work and the conclusion, respectively.

2 Model & Problem Statement

2.1 Basics of Distributed Query Plans

Common to other distributed stream work [2, 9], we assume the distributed stream processing system (DSPS) is deployed on a shared-nothing homogeneous compute cluster connected by a high bandwidth network. Hence, network bandwidth is not our key bottleneck. The DSPS accepts a continuous $query$, an expression describing the user's information needs. Then it performs a two-step optimization [8], namely, plan generation and operator placement, to determine the most effective strategy to execute the given query. Plan generation identifies the algebra operator plan with the least estimated cost for a given input query and estimated data stream statistics. Operator placement takes the algebra plan as input and outputs a mapping of each operator in the algebra plan to a physical machine (node) in the cluster. We refer to the algebra plan and its operator placement as *logical* and *physical plans*, respectively.

2.2 Multi-dimensional Parameter Space

Current optimizers [2, 9] tend to use a single-point statistic estimate for plan generation. However, it is well known that estimates in streaming environments tend to fluctuate over time. We thus now model these uncertainties in estimates via a multi-dimensional space around these estimates, called *parameter space* S. This space captures all possible combinations of estimate variations. Each point pnt in space S is a vector $< d_1, ..., d_n >$, where each d_i is an estimate of the corresponding statistic modeled by that dimension of the space such as selectivity or input rate.

```
**Algorithm 1** Compute the parameter space for $E$
**Input:** $E$-Statistic Estimates, $U$-Uncertainty Level
**Output:** $E_{lo}$, $E_{hi}$
  $\Delta = 0.1$ //Unit step
  **for** $i = 1 \rightarrow E.size()$ **do**
    $E_{hi}[i] = E[i] \times (1 + \Delta \times U)$;
    $E_{lo}[i] = E[i] \times (1 - \Delta \times U)$;
  **end for**
  **return** $E_{lo}, E_{hi}$
```

Different methodologies for constructing a parameter space exist, including strict upper and lower bounds [10] or levels of uncertainty to the optimizer estimates [11]. We use the latter approach in which the uncertainty level U is computed based on how statistic estimates E are derived. For example, if a value of E is available from the representative training data set, then $U = 1$ denotes low uncertainty. For simplicity we henceforth use an integer domain to denote the uncertainty levels, though other scales of uncertainty could be easily plugged in. The most crucial statistic estimates E for query optimization are the selectivities of operators and the input rates of streams [10]. We assume the statistic estimates E and the uncertainty level U correctly represent the data stream fluctuations. If suddenly some totally unexpected fluctuation arises in the future, our current solution may not be able to handle it, and we may have to exploit operator migration to resolve such scenarios after all. The parameter space S is computed as shown in Algorithm 1 and Example 2.

Example 2. Assume a simplified query $Q2$ of $Q1$:

```
Q2:  SELECT *
FROM News as N, Stocks as S, Currency as C
WHERE N.subject = S.industry        /*op1*/
AND N.country = C.country           /*op2*/
WINDOW 60 seconds
```

Assume neither the selectivity for operator op_1 nor the input rate of stream News are accurate over time due to data fluctuations (e.g., breaking news in a certain industry). To capture this uncertainty, a parameter space is constructed around the single-point estimate $E = \{\delta_1 = 0.4, \lambda_N = 100 \text{ tuples/sec}\}$. First, an uncertainty level U (e.g., $U = 2$) is assigned to each estimate in E. Then the parameter space is constructed with δ_1 ranging between 0.32 and 0.48, and λ_N ranging between 80 tuples/sec and 120 tuples/sec. An example of the parameter space is depicted in Figure 3.

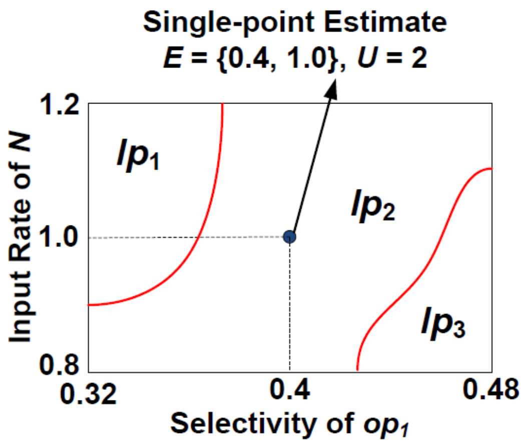


Figure 3. Selectivity Space Example

Similar to parametric queries [12], in practice, each dimension of the parameter space is discretized. For ease of exposition, we henceforth work with a 2D parameter space, though the extension to higher dimensions is straightforward.

2.3 Notion of Plan Robustness

Let us now consider the most common queries, namely, select-project-join (SPJ) queries. Thus the cost model of a logical plan in a 2D parameter space is of the form:

$$cost(p, pnt) = c_1 \cdot \sigma_i + c_2 \cdot \sigma_j + c_3 \cdot \sigma_i \cdot \sigma_j + c_4$$

where c_1, c_2, c_3, c_4 are coefficients, and σ_i, σ_j represent the selectivities of operators op_i and op_j, respectively. Modeling a specific plan requires suitably choosing the four coefficients. This is achieved through standard surface-fitting techniques [12]. Extending the above equation to a general n-dimensional space is straightforward. Given the notations in Table 1, we introduce the notion of robust query processing.

Table 1. Notations and Definitions

Term	**Definition**
n_i	The ith node
op_i	The ith operator
δ_i	The selectivity of op_i
r_i	Resource limit on node n_i
S	Parameter space
pnt	Parameter instance in the parameter space
pnt_{Hi}	Right top corner of a parameter space
pnt_{Lo}	Left bottom corner of a parameter space
lp	Logical query plan
LP_i	Robust logical solution for S, a subset of LP
pp	Robust physical plan
$cost(lp, pnt)$	Cost of a query plan p at pnt
$cost(OP_i)$	Cost of a subset of OP on the ith node (i.e., the workload on the ith node
lp_{pnt}^{OPT}	Optimal query plan at pnt

Definition 1. Given a parameter space S_i, a logical plan lp is $\epsilon - robust$, also called *robust logical plan*, in S_i if its costs satisfy the condition (see Table 1 for notions):

$$cost(lp, pnt_{Hi}) \leq (1+\epsilon) \times cost(lp_{pnt_{Hi}}^{OPT}, pnt_{Hi})$$

The intuition of what a robust logical plan is can be explained as follows: consider two optimal query plans lp_1 and lp_2 for two points $pnt_{Lo} =< d_{1,1}, ..., d_{1,n} >$ and $pnt_{Hi} =< d_{2,1}, ..., d_{2,n} >$ in the parameter space S_i respectively, and $pnt_{Lo} < pnt_{Hi}$, meaning $\forall i \; d_{1,i} \leq d_{2,i}$. If lp_1 is a robust plan in S_i, then we can provably bound the costs of plan lp_1 between the costs of plan lp_1 at pnt_{Lo} and the costs of plan lp_2 at pnt_{Hi}.

Definition 2. The *robust region* of a logical plan lp in S is the subarea S_i where lp satisfies Definition 1 at all points $pnt_i \in S_i$. For example, the robust region of lp_1 in Figure 4 (left) covers the left-top corner of the space S.

Definition 3. Given a set of robust logical plans, also called robust logical solution LP_i and resources r_i for node n_i ($\forall i : 1 \leq i \leq N$) for a DSPS, a physical plan pp is $robust$, also called *robust physical plan*, if it satisfies the following conditions: 1) $cost(OP_i) \leq r_i$, 2) $\bigcup OP_i = OP$, and 3) $\bigcap OP_i = \emptyset$ ($\forall i : 1 \leq i \leq N$), where OP_i denotes the set of operators allocated to machine n_i by pp, and OP is the full set of operators in the query.

Intuitively, a physical plan pp, shown in Figure 4 (right), is robust, if for each subset of query operators OP_i assigned to node n_i, its total cost $cost(OP_i)$ is no greater than the resource capacity r_i of n_i to execute the sub-plans of all query plans $lp_i \in LP$. Each OP_i associated with n_i also defined as a $configuration$ c_i, has no overlap with any other OP_j, and the union of all OP_i forms the whole operator set OP.

2.4 Problem Statement

Robust distributed query processing aims to 1) identify a robust logical solution LP_i, such that for each point in the parameter space S, there is at least one logical plan lp_i in the solution LP_i that is ϵ-robust by Def. 1 in that sub-space S_i, and to 2) produce a physical plan pp that supports the robust logical solution, i.e., pp is robust by Def. 2. The key idea is that the resulting system will be able to withstand the known data stream fluctuations, meaning, as long as the actual statistics (i.e., input stream rates and selectivities) remain within the parameter space. Our robust load distribution (RLD) problem can be formalized as follows.

Given a query q, resources r_i for node n_i ($\forall i : 1 \leq i \leq N$), statistic estimates $E =< e_1, \cdots, e_k >$, and the associated uncertainty levels $U =< u_1, \cdots, u_k >$, the ***robust load distribution problem*** *is to find a robust physical plan pp that supports a robust logical solution LP_i in the parameter space S constructed based on E and U.*

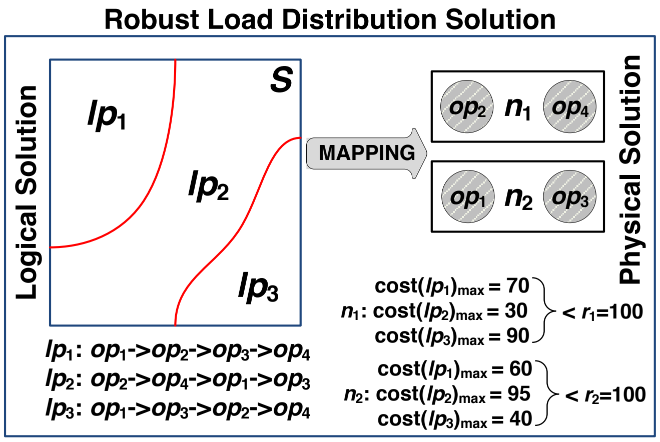


Figure 4. Robust Load Distribution Solution

Finding the robust solution (Figure 4) for this problem requires a comparison among all possible subsets of all logical plans LP and all possible physical plans pp in PP in the worst case [9]. The corresponding search space is a combined space of all possible robust logical solutions (sets of logical plans that together "cover" the fluctuations in streams), and the robust physical plan space that provides a static allocation of operators to machines so that this allocation can support a given robust logical solution. Therefore, the above problem is prohibitively expensive as the search space of finding robust logical and physical plans are exponential in the number of query operators and the number of machines in the system, respectively [9]. Furthermore, finding a robust logical solution LP_i and a corresponding physical plan pp supporting LP_i requires the optimizer to search both logical and physical search spaces. This renders the solution intractable for large problems, e.g., large numbers of query operators or machines.

3 Overview of RLD Approach

Given the intractability of RLD, we instead employ a two-step approach towards query optimization popular for both distributed and parallel database systems [8] due to its reduction of the overall complexity of the distributed query optimization problem.

In our context, the two-step optimization works as follows: 1. The first step generates a robust logical solution, in which each logical plan is designed for a particular sub-region of the parameter space. 2. The second step produces a single robust physical plan, determining the machine on which each operator is placed and supporting the given logical solution without load redistribution.

The above two steps efficiently work together to achieve an effective load distribution plan as our experiments confirm. The first step realized by our proposed ERP approach, described in Section 4.3, has a much smaller optimization complexity than parametric query optimization [13, 14, 15]. The second step reduces the complexity of producing a physical plan by prioritizing robust logical plans to support. Our overall RDL strategy conducts load distribution with full awareness of all possible scenarios at runtime, thus avoiding costly load redistributions.

We now briefly introduce the key challenges tackled by our RLD: 1. How do we efficiently produce a robust logical solution LP_i that covers the complete parameter space (Step 1)? 2. Can we produce a single robust physical plan pp that satisfies all or at least the maximal number of robust logical plans in LP_i (Step 2)? Intuitively, our strategy exploits a probabilistic model to guarantee that any missing robust logical plans cannot occupy a large area in the space. Thus any uncovered area has a probability below some bounded threshold. The RLD architecture is depicted in Figure 5.

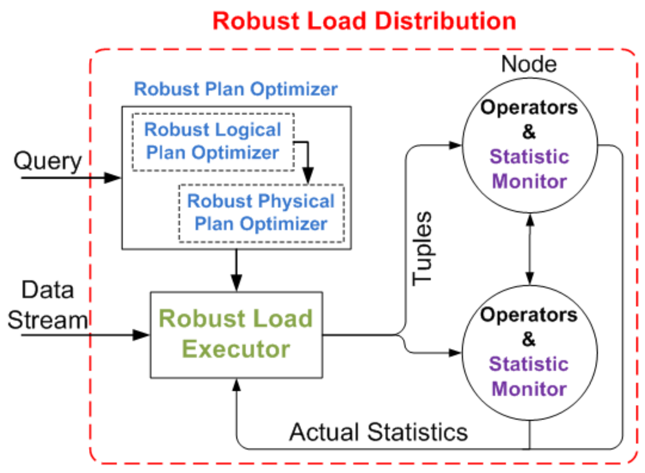

Robust Load Distribution
Node
Robust Plan Optimizer
Robust Logical Plan Optimizer
Robust Physical Plan Optimizer
Query
Data Stream
Robust Load Executor
Tuples
Operators & Statistic Monitor
Operators & Statistic Monitor
Actual Statistics

Figure 5. Robust Load Distribution Architecture

Robust plan optimizer. Our robust plan optimizer uses the standard query optimizer of a DSPS as a black-box to perform traditional plan optimization calls. Given a query, it estimates resource requirements of the query based on the uncertainty of the estimates. The plan optimizer then produces a *robust logical solution*, and determines a single robust physical plan to support this logical solution. The remainder of this paper details this optimizer.

Robust load executor. After a robust physical plan is selected, the operators are instantiated by the executor on the machines accordingly. At runtime, our executor collects up-to-date statistics of running operators from statistic monitors installed on the DSPS machines. The executor assigns the most appropriate logical plan from the robust logical solution to the new arriving tuples based on the latest runtime information. To keep the costs of making plan decisions minimal, a logical plan is assigned to tuples in batches. Techniques from the literature on multiple-route query processing [16, 17] are plugged in for runtime plan execution.

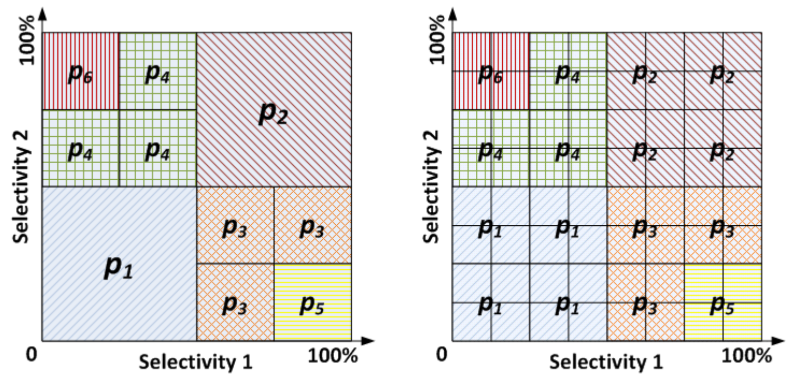
In particular, RLD is built on top of the QueryMesh executor [16] that offers the ability of switching between robust logical plans at runtime by an online classifier operator. Once a robust load distribution solution has been produced by the optimizer, the online classifier is implemented that associates each computed robust logical plan with the specific statistics. The classifier then inspects the latest runtime statistics to determine the best logical plan from the robust logical solution for each batch of tuples. In the QueryMesh infrastructure, each batch of tuples then carries its own execution path (i.e., its own logical plan ordering). This is agile, as it is in fact always the system to adapt between logical plans without any migrations.

Statistic monitor. The robust load executor requires runtime knowledge about the actual values of key parameters. Thus each machine in a DSPS runs a statistic monitor that periodically samples the selectivity of operators and the associated stream input rates. The monitor then transmits the statistics to the executor where all statistics are kept up to date.

4 Robust Logical Plan Generation

4.1 Weighted Partition Algorithm Overview

The parameter space, defined in Section 2, requires us to find the robust logical solution LP_i such that each plan $lp_i \in LP_i$ satisfies Def.1. The exhaustive approach would incur impractically large computational overhead for high-dimensional spaces. Random sampling also suffers from potentially huge computational costs and tends to fail to identify appropriate robust logical plans in the space, as confirmed by our experiments in Section 6.3. Instead our approach leverages the fact that we are not required to identify the *optimality region* for each plan, but rather only its *robustness region* (as per Def. 2). As our experimental study validate, this reduces the space significantly.



a) Ideal approach b) Exhaustive approach

Figure 6. Comparison of Ideal and Exhaustive Approach

Consider a two dimensional parameter space containing 6 distinct robust logical plans, with their respective robust regions depicted in Figure 6(a). In this example, the parameter space would only need to be partitioned twice (making 10 optimizer calls) to confirm the robustness of all 6 plans. The resulting partitioned space will contain 10 sub-spaces (Figure 6(a)). On the contrary, the exhaustive approach depicted in Figure 6(b) would divide the parameter space into an $8\times$ 8 grid (making 64 optimizer calls) to discover the same 6 plans. Thus, the exhaustive approach is 6 times more expensive than needed. Worst yet, such overhead increases significantly with the number of dimensions of the parameter space.

Overview of the Logical RDL Steps. The first technical challenge addressed by our logical plan generation algorithm is how to efficiently find an effective partition point in the parameter space (Step 1). By the robust logical plan definition, to verify the robustness of a logical plan, we must make expensive optimizer calls in all sub-spaces of its partition. If that tested plan does not satisfy the robustness criteria, a finer-grained partition must be found. Therefore, identifying a good partitioning point is the key to avoid repeated wasteful attempts of expensive robust logical plan finding. Our insight is to leverage information about the already known query plans in the space and encode this knowledge as weights in the space. In this model, points where a new robust plan is more likely to exist are assigned higher weights.

The second issue we tackle is how to update the weights assigned to all points during the parameter space partitioning process (Step 2). Given the size of the space, we propose an efficient approach that incrementally updates the weights.

The third issue we address is when to stop partitioning the parameter space (Step 3)? Fine-grained partitioning incurs significant computational overhead for query optimization. Moreover, the quality improvement of the resulting robust logical plans may no longer outweigh its growing expense of optimizer calls. Thus, we develop a termination condition that provides a probabilistic guarantee on the space coverage by the robust logical plans. We show that the possibly missed robust plans, if any, are guaranteed to not occupy a large area in the parameter space.

While on the surface our problem relates to parametric query optimization, our approach of generating robust logical plans has several crucial differences from generating plan diagrams in parametric query optimization (i.e., finding all optimal plans in a given parameter space) [12, 18]. Detailed comparison can be found in Section 7. The efficiency and effectiveness of our proposed strategy is experimentally confirmed in Section 6.3.

4.2 Weight Assignment in Parameter Space

We now describe our strategy of exploiting plan cost information from already identified plans to assign weights to the remaining points in the parameter space. The weight of a point should reflect the probability that the robust logical plan at that point is different from the robust plans in the bottom-left and top-right corners of the associated space. This way we would be able to exploit these weights to efficiently identify distinct robust plans in the parameter space. However, this is clearly infeasible without knowing what the costs of previously identified plans would be at all points in the space. Yet computing all these plan costs is prohibitively expensive in a large space. Thus, our goal instead is to heuristically approximate the weights without exhaustively computing all plan costs in the space. For this, we develop a weight function based on the following principles.

Principle 1. Two points in the parameter space that are closer to each other are more likely to have the same robust plan compared to a more distant pair of points. To motivate this, consider the cost model in Section 2.3. We observe that the cost of a plan is monotonically increasing along each dimension of the space [12]. Thus, in a small region of the space, the costs of a plan lp_i on two nearby points are likely to have the same robust plan by Def.1.

Principle 2. A plan is less likely to be robust at a point if the slope of the plan's cost function, defined in Section 2.3, at that point is high. This principle is based on the observation that the slope of a plan's cost function is monotonically increasing in the parameter space. Intuitively, the closer the points move towards the margin of the plan's robust region, the higher the slope of the plan cost function is at these points.

Based on these two principles, we now design a **weight assignment function** that increases as the ratio between the slope of a plan's cost function and the distance of that point to the bottom left point pnt_{Lo} of that space increases. The weight function decides how quickly the weight decreases as the distance increases, and how quickly the weight increases as the slope increases. In practice, assigning weights individually to each point in the parameter space would be expensive since there are $O(n^d)$ points in a d-dimensional space assuming an n step discretization of the space along each dimension. Thus, we consider each dimension independently. A point $pnt = (x_1, \cdots, x_n) \in S$ is projected onto each dimension d_i, such that $x_i \in d_i$ and the point's weight on each dimension is assigned according to the projected distance between pnt and pnt_{Lo} (see Table 1). The weight of pnt in the i-th dimension, denoted by $weight_i(pnt)$, is defined as:

$$weight_i(pnt) = \frac{min(slope(pnt, p^{OPT}_{pnt_{Hi}}), slope(pnt, p^{OPT}_{pnt_{Lo}}))}{dist(pnt, pnt^{i}_{Lo})}$$

where $dist$ is a function that measures the distance between two points. Any distance measure such as Manhattan or Euclidean Squared Distance [19] could be plugged in.

Weight Re-Assignment Strategy. Unfortunately, the initial weight assignment will no longer be accurate after partitioning. That is partitioning produces several sub-spaces, each of which may have their own optimal plan at bottom-left $lp^{OPT}_{pnt_{Lo}}$ or top-right corner $lp^{OPT}_{pnt_{Hi}}$ of that sub-space. Recall that the optimizer finds a distinct robust plan for each sub-space. Thus, each point's weight has to be updated accordingly in order to reflect the cost behavior of the new plans in the sub-spaces.

As described earlier, the cost of the weight assignment function largely depends on the number of points in the parameter space. Updating the entire parameter space repeatedly incurs significant overhead for the weight assignment. We now introduce a refinement of the above weight assignment approach where we update the weights of points in a sub-space as we partition the parameter space S if and only if the following condition is satisfied.

$$\overline{(lp_{pnt_{Lo}} = lp^{OPT}_{pnt_{Lo}}) \wedge (lp_{pnt_{Hi}} = lp^{OPT}_{pnt_{Hi}})} = \text{True}$$

where $weight^{'}_{i}(pnt)$ denotes the updated weight assignment for the current partition point pnt; while $S_i.pnt_{Hi}(S_i.pnt_{Lo})$ denotes the top-right (bottom-left) corner of the sub-space $S_i \in S$. Intuitively, the above condition ensures that the update would not be triggered if the predicted logical plan at top-right (bottom-left) corner is identical to the optimal plan identified at the same point after partitioning the space.

Example 3. We illustrate the above weight re-assignment strategy using Figure 7. The weight is assigned to each point in the original parameter space in Figure 7(a). Then we partition the space into 4 sub-spaces, and compute the weights of pnt_{Hi} and pnt_{Lo} for each of the 4 sub-spaces S_i (Figure 7(b)). However, we only update the sub-spaces S_4 (Figure 7(c)) as the optimal plan is different from the predicted robust logical plan at $S_4.pnt_{Hi}$. Thus, the above condition reduces the number of sub-spaces to update.

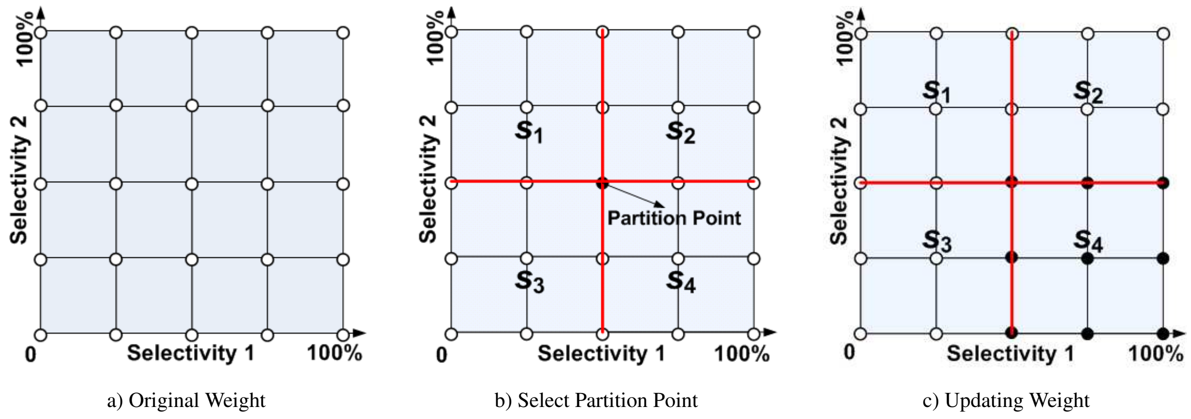


Figure 7. Weight Assignment for Parameter Space

Example 3 shows savings by not updating weight assignments in areas where an identified robust logical plan has a ”large” area of optimality. The reason is that the weight assignment in a sub-space only depend on its robust plan. Thus, we avoid re-computations of any weight assignments that is constant in the area.

4.3 Parameter Space Partitioning

4.3.1 Weight-driven Robust Partitioning Algorithm

We now illustrate how the above weight assignment is integrated into the parameter space partitioning process. The main idea of parameter space partitioning is to incrementally find *robust* plans as the space is partitioned. A straightforward way to achieve this is using the *weight-driven partition (WRP)* procedure (Algorithm 2). First we compute the weights for all points in the discretized space S. Then, we pick the point with the highest weight as the partition point to divide the space into 2^d sub-spaces, with d denoting the number of dimensions of S. Each sub-space S_i will be further partitioned if its optimal plan at pnt_{Hi} is not *robust*. Finally, the partitioning process stops once all plans at pnt_{Hi} of all S_i are *robust*.

This partitioning process could result in unnecessarily small sub-spaces if the robustness threshold ϵ is not well chosen. On the contrary, a too large ϵ may end up with a rather small number of sub-spaces - leading to suboptimal plans. Our experiments (Section 6.3) show that relatively small increments in ϵ are sufficient to bring down the number of plans significantly, without adversely affecting the query processing quality.

Limitations of WRP. As described above, *WRP* aims to minimize the computational overhead of partitioning. Yet there is a significant explosion in costs associated with an increasing dimensionality of the parameter space. Moreover, a large percentage of the computation could be wasted as WRP treats all sub-spaces in the parameter space equally rather than targeting specific subareas. The intuition is that a too $strict$ threshold ϵ may lead the partitioning algorithm to undertake a too fine-grained job, not merited by the coarseness of the underlying space. In an extreme case, all optimal logical plans in the space must be identified if ϵ = 0.

Algorithm 2 Weight-driven robust partitioning algorithm

```
Input: A parameter space $S$
Output: A robust logical solution $LP_i$
 1: Plan $lp \leftarrow$ optimize($S.pnt_{Hi}$)
 2: if $lp$ is $robust$ in $S$ then
 3:     $LP_i.add(lp, S)$
 4:     return $LP_i$
 5: else
 6:     Assign weights to points in $S$;
 7:     Choose appropriate point to partition $S$ based on weights
 8:     for $i = 1 \rightarrow n(numberofsub - spaces)$ do
 9:         Update weight assignments in $s_i \in S$
10:         $standardPartitioning(s_i, LP_i)$;
11:     end for
12: end if
13: return $LP_i$
```

4.3.2 Early-terminated Weight Robust Partitioning

Given the shortcomings of *WRP*, we now enhance this approach by designing an early termination strategy. This new method, called ERP, reduces the overhead while providing *a probabilistic guarantee that the robust plans missed cannot occupy a large area in the parameter space*. Our solution uses two key factors:

(a) **Region of robustness** for a given plan lp refers to the location where the plan lp is robust in the parameter space.

(b) **Size of robustness** for a given plan lp corresponds to the total area in the parameter space where the plan lp is robust.

We exploit the above two factors to trade off between the partitioning costs and the quality of the resulting plans. Observe that when we partition the parameter space using uniform partitioning, the probability of finding a new robust plan is proportional to its area of robustness. Given that a finite number of robust plans exists, the probability of finding a new robust plan decreases as we find more robust plans from that set of plans when partitioning. The more robust plans we have already found, the more partitioning steps it will take on average to find an additional robust plan. We propose to exploit this insight by terminating the partitioning process when we have not obtained a new robust plan for a pre-determined number of partitioning steps.

For this, we maintain an *aging counter*, which keeps track of the interval between the last two times that a new robust plan was detected in the partitioning process. Each time we call the optimizer at pnt_{Hi} of a new sub-space, if the plan at pnt_{Hi} is a new robust plan that is distinct from all robust plans observed thus for, we reset the aging counter. Otherwise, we increment the aging counter.

Assume that after partitioning t times, we found a new robust plan. Then the aging counter is reset to 0 and we start counting up again. Let c be the number of additional partitioning steps after t to find the next new robust plan. The probability of finding a robust plan is constant between t and t + c, since the number of missing robust plans does not change during this interval. If we denote the probability of identifying a new robust plan after t partitions by $Pr(t)$,

$$\forall\, 0 \leq t \leq c\text{ - }1,\, Pr(t + c) = \frac{n_{miss}}{n_{total}}$$

where n_{miss} is the number of unidentified robust logical plans and n_{total} is the total number of robust plans.

Theorem 1. *With a probability of at least 1 - ε, if we do not find a new optimal plan in the partitioning process within c trials, where c $\leq$ c_0 = (1 + $\varepsilon^{-1/2}$) / δ, then the total number of optimal plans not yet found is bounded by δ ($\leq \delta$).*

Proof. Let p be the probability of finding a new robust plan. Then the expected value E[c] = 1/p, and variance Var[c] = $(1-p)/p^2$. By Chebyshev's inequality [20], for any $k \in \mathbb{R}^+$,

$$Pr\left[|c - \frac{1}{p}| \geq k\sqrt{\frac{1-p}{p^2}}\right] \leq \frac{1}{k^2}.$$

We wish to bound p using the above inequality. Solving the inside for p gives,

$$c^2 - \frac{2c}{p} + \frac{1}{p^2} \geq k^2 \frac{1-p}{p^2}$$

$$c^2p^2 + (k^2 - 2g)p + 1 - k^2 \geq 0$$

Two solutions of the above inequality are:

$$p_1 = \frac{2c - k^2 - k\sqrt{k^2 + 4c(c-1)}}{2c^2}$$

$$p_2 = \frac{2c - k^2 + k\sqrt{k^2 + 4c(c-1)}}{2c^2}$$

Expressing the bound using p, p_1 and p_2,

$$Pr[p \leq p_1 \vee p \geq p_2] \leq \frac{1}{k^2} \Rightarrow Pr[p \geq p_2] \leq \frac{1}{k^2}$$

Let $c_0 = (1+k)/\delta$. If $c \geq c_0 = (1+k)/\delta$ then $\delta \geq (1+k)/c$. From $\delta \geq p_2(\delta \geq (1+k)/g > p_2)$, $Pr[p \geq \delta] < Pr[p \geq p_2] \leq 1/k^2$. Setting $\varepsilon = 1/k^2$ proves the theorem. □

Intuitively, Theorem 1 states that if the aging counter reaches a value $\geq c_0$, then with high probability the missing optimal plans cover a small area in the parameter space. We observe that the position to partition the space can have a significant impact on how quickly the aging threshold is reached. The space partitioning technique, which exploits Theorem 1 to early terminate the partitioning process, called ERP, is shown in Algorithm 3. Since the guarantee in Theorem 1 is probabilistic, our proposed ERP may miss some robust plans. The following theorem quantifies the likelihood of missing a robust plan based on its area of optimality.

Theorem 2. *Suppose we stop partitioning according to the aging threshold in Theorem 1. If the coverage of an optimal plan lp is $area(lp) \geq \gamma \cdot \delta$, for a constant 0 < γ $\leq$ 1/δ, then the probability that the plan lp is not found is at most $e^{-\gamma(1+\varepsilon^{-1/2})}$.*

Algorithm 3 Early-terminated Robust Partitioning Algorithm

```
Input: A parameter space $S$
Output: A robust logical solution $LP_i$
1: $LP_i \leftarrow \phi$
2: $counter_{miss} \leftarrow 0$
3: $age\_threshold \leftarrow (1 + \varepsilon^{-1/2})$ / $\delta$
4: while $counter_{miss} \leq age\_threshold$ do
5:     $pnt \leftarrow getPartitionPnt(S)$
6:     $p_{pnt}^{OPT}$ is the optimal plan at $pnt$
7:     if $p_{pnt}^{OPT} \in LP_i$ then
8:         $counter_{miss}$++ and continue
9:     else
10:        $add(p_{pnt}^{OPT})$ to $LP_i$
11:        $counter_{miss}$ = 0
12:    end if
13: end while
14: return $LP_i$
```

Proof. Assume that the total number of robust plans is $|P|$ and the total area of the selectivity space is A. Consider the probability of missing a robust plan $p.log$ with $area(p.log) \geq \gamma \cdot \delta A$, denoted $Pr[p.log \notin P.log]$. γ is a constant such that 0< $\gamma \leq$ 1/ε. Since $r \geq c_0$ and $1 - \frac{area(p.log)}{A} \leq 1$,

$$Pr[p.log \notin P.logs] \leq (1 - \frac{area(p.log)}{A})^{|}P.log| \leq (1 - \frac{area(p.log)}{A})^{c_0} \leq$$

$$(1 - \gamma\delta)^{c_0} \leq e^{-\gamma\delta \cdot \frac{1+\varepsilon^{-1/2}}{\delta}} = e^{-\gamma(1+\varepsilon^{-1/2})}.$$

□

Theorem 2 states that if an optimal plan has a "large" area of optimality, then we will find it with high probability when using the stopping condition of Theorem 1. From Theorem 2 we know that the probability of missing a robust plan decreases exponentially with its area. *Thus, while Theorem 1 refers to the total uncovered area due to all missing robust logical plans, Theorem 2 confirms that individual robust plans with any non-trivial area will be found with high probability.*

5 Robust Physical Plan Generation

5.1 Basic Approach for Robust Physical Plan

A straightforward strategy for robust physical plan generation would entail the following steps. As input, we accept LP_i, a set of logical plans that together cover the parameter space by Def.1 produced by the logical optimizer. Then we compute all physical plans PP for each robust plan $lp_i \in LP_i$, denoted by $PP(lp_i)$. Thereafter, we find the intersection among all sets $PP(lp_i)$ for all logical plans $lp_i \in LP_i$. If the intersection is not empty, then any physical plan in this intersection is a valid solution, which satisfies all robust logical plans in LP_i.

However, if the intersection is empty, then no physical solution supports *all* logical plans in LP_i. We would need to locate a suboptimal solution. Many strategies are possible. One simple option would be to remove a logical plan lp_i from solution LP_i. Then we repeat the above procedure until the intersection is not empty and thus a valid robust physical plan can be produced.

Unfortunately, the number of physical plans for a single logical plan is $n^m/n!$ [9], where n is the number of machines and m the number of operators in the logical plan lp. Moreover, the number of different logical solutions LP_i is $2^k - 1$, where k is the cardinality of the set of all possible logical plans LP. As a result, finding the optimal solution for this problem is intractable for a large number of machines or a large number of robust logical plans.

Thus, we now propose algorithms that trade off between the optimization complexity versus the result optimality. $GreedyPhy$ exploits heuristics to efficiently find a robust physical plan in polynomial time, whereas $OptPrune$, using $GreedyPhy$ as baseline, is guaranteed to find the optimal physical plan to support the maximum number of logical plans, though with minimal increase in optimization time.

5.2 Greedy Physical Plan Generation

Given the complexity of physical plan generation, we now introduce a heuristic-driven strategy that uses two key principles:

(a) ***The area of optimality heuristic***. Intuitively, we aim to produce a robust physical plan pp that covers as much as possible of the parameter space by supporting all logical plans in LP_i. If all logical plans cannot be supported by pp, then we drop from LP_i the least important logical plan, i.e., the plan associated with the *smallest robust region* in the parameter space S from LP_i.

(b) ***The probability of occurrence heuristic***. By definition, the selectivity of an operator fluctuates around its given statistic estimates. Various probability distributions could be used to model the occurrence of a point in the space S. For simplicity, we model this probability using a *normal distribution* as commonly done in the literature [19]. Therefore, the closer a point is to the given statistic estimate, the higher the possibility that the actual selectivity happens at runtime. As a result, we drop the logical plan whose optimality region is the furthest away from the given estimate point.

Weight Assignment Policy. Our strategy is to assign a weight to each robust logical plan that incorporates the above two factors. Let $area(lp_i)$ denote the robust region of plan lp_i, and $Pr(pnt_j)$ denote the probability of occurrence of a point pnt_j at run time. A robust logical plan's weight $weight(lp_i)$ is defined as:

$$weight(lp_i) = \sum_{pnt_j \in area(lp_i)} Pr(pnt_j)$$

where $Pr(pnt_j)$ can be obtained from the normal distribution.

Example 4. Consider a 2-dimensional parameter space with each dimension discretized into 16 units (see Figure 8). Suppose the space contains 5 different robust logical plans. Each plan's robust region is depicted as one or more rectangles. For example, the robust region of lp_1 includes $area_1$, $area_6$, $area_{11}$, and $area_{16}$. The probabilities of the actual run time statistics to fall within these rectangles are 2.4%, 11.7%, 11.7%, and 2.4%, respectively. Summing the values gives us the weight of lp_1 as 28.2%. The probability of occurrence with respect to a unit area is calculated as follows using $area_1$ as an example:

$$Pr(area_1) = Pr_x(area_1) \cdot Pr_y(area_1)$$

where x and y denote the x-axis and y-axis of a unit area in the 2-dimensional parameter space, respectively. The above assumes that the x and y dimensions are independent following the assumption of independence of selectivity values commonly made by query optimizers [21]. Thus the correlation between dimensions is zero.

Example 5. $Pr_x(area_6) = Pr_x(0.3 \leq x \leq 0.5) = Pr_x((0.3 - \mu)/\sigma \leq Z \leq (0.5 - \mu)/\sigma)$, where μ is the mean (the estimated selectivity) and σ is the standard deviation (the uncertainty level of the estimated selectivity) of the selectivity in x-axis. For example, in Figure 8, if $\mu = 0.5$ and $\sigma = 0.2$, so $Pr_x(area_6)$ = 0.342. Similarly, $Pr_y(area_6)$ = 0.342. Multiplying $Pr_x(area_1)$ by $Pr_y(area_1)$, we get $Pr(area_1)$ = 0.117. Finally, $weight(lp_1)$ = $Pr(area_1) + Pr(area_6) + Pr(area_{11}) + Pr(area_{16})$ = 0.282.

The GreedyPhy Algorithm. After assigning weights to the logical plans, the plans are stored in a heap sorted by their weights in descending order. $GreedyPhy$ algorithm exploits the above weight assignment (Algorithm 4).

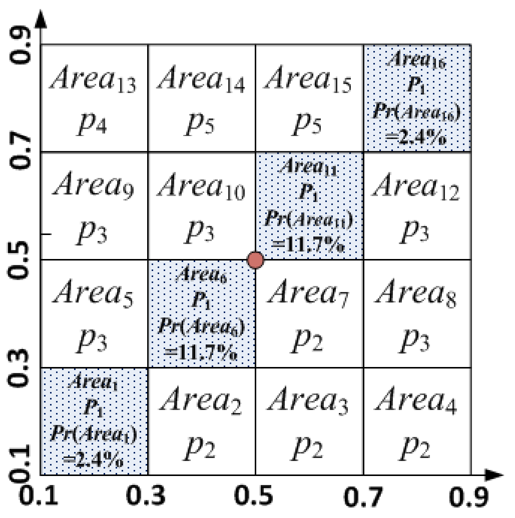


Figure 8. Weight Assignment for Logical Plans

Given a solution LP_i produced by the logical plan optimizer, then each plan $lp_i \in LP_i$ has a $weight$ $w(lp_i)$, $GreedyPhy$ finds the physical plan pp that supports a subset of LP_i such that the sum of all weights of the supported logical plans is maximal among all possible subsets of LP_i. Intuitively, the resulting robust physical plan pp should be robust to the most frequently occurring data fluctuations. Given a robust logical solution LP_i and resource constraints r_i, $GreedyPhy$ generates a logical plan lp_{max}, in which the cost of each operator is equal to its maximum cost for all logical plans $lp_i \in LP_i$ (Lines 1-2). Thereafter, in each iteration the algorithm tries to produce a physical plan by using the Largest Load First (LLF) algorithm (i.e., Longest Processing Time algorithm [9]) (Line 4). LLF orders the operators by their cost and assigns operators in descending order to the least loaded machine. If a physical plan is produced by LLF, it is thus returned as final solution.

If the algorithm fails to find a physical plan, it removes the least-weighted logical plan lp_i from the robust logical solution LP_i (Lines 8-10). After repeating lines 3-10 a maximum of $|LP_i|$ times (LP_i is empty after $|LP_i|$ times), the algorithm returns a physical plan that maximizes the total weight of the subset of logical plans selected from LP_i supported by pp.

Complexity Analysis. The worse case for $GreedyPhy$ is that none of the logical plans in LP_i can be supported by the given resources. In other words, it would iterate k (the cardinality of $|LP_i|$) times before stopping. Therefore, the worse case complexity of $GreedyPhy$ is $O(n)$ as the complexity of LLF is proven to be $O(n)$ [9], and the complexity of $getMinWeightPlanWithMaxOp$ and $updateMax$ procedures are both linear in the number of operators in lp, namely $O(n)$. Therefore, $GreedyPhy$ is guaranteed to have a linear complexity taking $O(n)$ time.

5.3 OptPrune Physical Plan Generation

The above algorithm, being greedy, cannot always find the optimal solution. We now design a strategy that guarantees the $optimal$ solution is found if one exists. Given the prohibitively high complexity of exhaustive search, the key idea is to devise a pruning methodology that eliminates suboptimal solutions. $OptPrune$ succeeds in improving the efficiency of the search costs without compromising result optimality (see Algorithm 5).

In order to find the optimal solution, $OptPrune$ potentially needs to examine all possible physical plans. We represent all physical plan candidates in a weighted directed graph $G = (V, E, SCORE)$ (Figure 9) where $v_i \in V$ are vertices, $e_{ij} \in E$ are directed edges $v_i \to v_j$, and a $score \in SCORE$ is associated with each vertex

Algorithm 4 GreedyPhy Algorithm

```
Input: A robust logical solution $LP_i$, resources $R$
Output: A robust physical plan $pp$
1:  $terminate \leftarrow$ false
2:  $lp_{max} \leftarrow updateMax(LP_i)$
3:  while $terminate \neq$ true do
4:      $pp \leftarrow LLF(lp_{max}, R)$
5:      if $pp \neq null$ then
6:          $terminate \leftarrow$ true
7:      else
8:          $index \leftarrow getMinWeightPlanWithMaxOp()$
9:          $LP_i.remove(index)$
10:         $lp_{max} \leftarrow updateMax(LP_i)$
11:     end if
12: end while
13: return $pp$
```

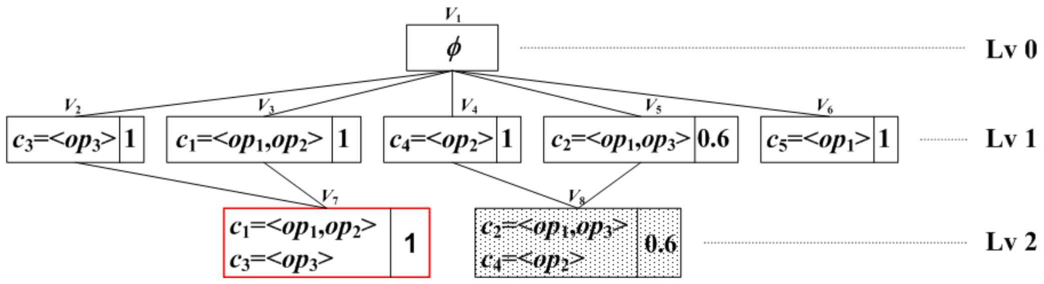


Figure 9. Robust Physical Plan Search Graph

v_i, denoted by $score(v_i)$. A vertex v_i represents a set of configurations c_i defined in Section 2.3. The $root$, the vertex on level 0, is empty. Each $leaf$, a vertex on level N (with N the number of machines), is a *robust physical plan* (Def.3). All vertices located between level 0 and level $N - 1$ correspond to partial physical plans (i.e., $\bigcup OP_i \sqsubset OP$ but $\bigcup OP_i \neq OP$). A vertex v_j is the $child$ of the vertex v_i denoted by the edge e_{ij} from v_i to v_j. The configurations in a child vertex v_i corresponds to the union of all configurations in v_i's parents. A $score$ is the minimal weights of the configurations can support in a vertex. For example, $c_2 = 0.6$ and $c_4 = 1$ in Figure 9, the score of $v_8 = min\{c_2, c_4\}$ = 0.6.

The key idea is that $OptPrune$ can leverage the results generated by $GreedyPhy$ as its pruning criteria for improved efficiency. For simplicity, we here assume that the machines are homogeneous. Thus a configuration such as $c_2 =< op_1, op_3 >$ is valid, if op_1 and op_3 can fit on one machine.

$OptPrune$ traverses the above search tree in a depth-first search (DFS). $OptPrune$ starts at the root, an empty plan, and continues iteratively down the graph forming a robust physical plan by adding configurations. As stated in Section 5.1, the goal of $OptPrune$ is to maximize the total score of the logical plans that a physical plan pp can support, denoted by $score(pp)$. The algorithm first figures out all possible configurations on a single machine (Line 1). The *score* of a physical plan is the total weight of the logical plans being supported. It sorts the configurations in decreasing order of the number of operators in each configuration. c_0 is the configuration with the most number of operators (Lines 2-6).

$OptPrune$ starts its depth-first search (DFS) by adding one configuration to the current robust physical plan (pp) at a time (Lines 7-11). If the union of configurations in pp contain all query operators, then the algorithm terminates and returns pp (Lines 12-13). We have an effective bound that is guaranteed to only eliminate suboptimal solutions. If the current partial pp exceeds the given resource capacity or the $score$ of the current pp is worse than that of the $GreedyPhy$ solution, then the algorithm backtracks by removing the newly added configuration from the current pp and updating the $score$ of pp accordingly (Lines 14-21).

Algorithm 5 OptPrune Algorithm

```
Input: A set of robust logical plans $LP_i$, resource limits $R$
Output: A robust physical plan $pp$
 1: $C \leftarrow$ all feasible configurations on a single machine
 2: $greedyPlan \leftarrow GreedyPhy(LP_i, R)$
 3: $bound \leftarrow score(greedyPlan)$
 4: $pp \leftarrow$ NULL
 5: $sort(C)$
 6: $c_0 \leftarrow max(C)$
 7: $Search(C, c_0)\{$
 8: $S' \leftarrow removeConflict(C, c_0)$
 9: for all $c_i \in S'$ do
10:    $pp.add(c_i)$
11:    $score \leftarrow updateScore(pp)$
12:    if $completeSolution(pp)$ then
13:       return $pp$
14:    else if $checkLimit(pp) \vee score < bound$ then
15:       $pp.remove(c_i)$
16:       $score \leftarrow updateScore(pp)$
17:       continue
18:    else if $!Search(C', c_i)$ then
19:       $pp.remove(c_i)$
20:       $score \leftarrow updateScore(pp)$
21:       continue
22:    else
23:       continue
24:    end if
25: end for
26: $\}$
27: return $pp$
```

Lemma 1. *In the search graph, when we add a distinct configuration to the current pp, the score of this newly constructed pp* is guaranteed to be no greater than that of the original pp.*

Proof. Assume a pp_i contains k configurations and the score of the pp_i is s_i. Now let us compare pp_i and its children (i.e., $k + 1$ configurations including pp_i's k configurations) on the next lower level of the search tree. Adding a configuration c_i to a partial physical plan pp either keeps or reduces the number of robust logical plans that the new physical plan supports. Thus the score of each child physical plan cannot be higher than that of pp. □

Theorem 3. *The new pp* cannot be an optimal solution if the current pp is not an optimal one. The physical plan search graph is guaranteed to be safely pruned. $OptPrune$ is guaranteed to find the optimal solution.*

Proof. Whenever the score of the current pp exceeds the $bound$, we can safely prune its branches because these vertices contain solutions that cannot become the final robust physical plan. □

Complexity Discussion. The worse case for $OptPrune$ is to have to check every configuration in the entire search space. Therefore, the worse case complexity of $OptPrune$ is the same as that of exhaustive search, namely, $O(n^m/n!)$. However, in practice our proposed pruning methods are found to be extremely effective at terminating the search much earlier, as confirmed by our experimental results (Section 6.4). The reasons are twofold. First, $GreedyPhy$ produces a relatively good physical plan - this offers us an excellent bounding condition, which enables us to stop searching through most branches early on. Hence it reduces the search space significantly without affecting the search accuracy. Secondly, $OptPrune$ terminates immediately if it finds the first physical plan (leaf) that supports all robust logical plans. Therefore, $OptPrune$ guarantees to produce a physical plan supporting either all logical plans or the most important logical plans within the available resource.

6 Experimental Study

We have implemented the proposed techniques on D-CAPE [22], a distributed continuous query processing architecture employing stream query engines over a cluster of shared-nothing processors. The experiments were run on the D-CAPE system using Linux machines with AMD 2.6GHz Dual Core CPUs and 4GB memory.

Table 2. System Parameters & Distribution Distribution

Parameter	Value	Description
Data Arrival	*Poisson*	Data arrival distribution
μ	500 msec	Mean inter-arrival rate
$\|T_{dq}\|$	1,000	Maximum # of tuples dequeued by an operator at a time
Ruster size	100 tuples	Minimum *ruster* size
Data Distributions		
Uniform (α = 0, β = 100): *min*: 0.0, *max*: 100.0, *med*: 49.0, *mean*: 49.7, *ave.dev*: 25.2, *st.dev*: 29.14, *var*: 849.18, *skew*: 0.05, *kurt*: -1.18.		
Poisson (λ = 1): *min*: 0.0, *max*: 7.0, *med*: 1.0,*mean*: 0.97, *ave.dev*: 0.74, *st.dev*: 1.01, *var*: 1.02,*skew*: 1.17, *kurt*: 1.89		

6.1 Data Sets and Queries

Stocks-News-Blogs-Currency data set: We have employed a data polling application, which is implemented in QueryMesh [16], to collect NYSE stock prices, foreign currency exchange rates from Yahoo Finance, news and blogs via RSS feeds.

Sensor data set: This data set contains readings from sensors in the Intel Research, Berkeley Lab [10]. The sensor readings are streamed to D-CAPE in the order they are generated, as if they were submitted by sensors in real-time.

Synthetic data sets: To study the effectiveness of our strategies under data fluctuations, we design several data sets using various data distributions that model real-life phenomena. Default properties, distribution and system parameters are depicted in Table 2.

Queries: We deploy N-way join queries, as those are among the core and most expensive queries in database systems. The default settings used in our experiments are listed in Table 2. The queries are equi-joins of 10 streams. The sliding windows are based on the application timestamps associated with the data (as opposed to the clock times when tuples arrived at the system during a particular test run). This ensures that the query answers are the same regardless of the rate at which the data set is streamed into the system or the scheduling of tuple processing (i.e., repeatable workloads).

6.2 Experimental Methodology

Our experiments are categorized into three major classes.

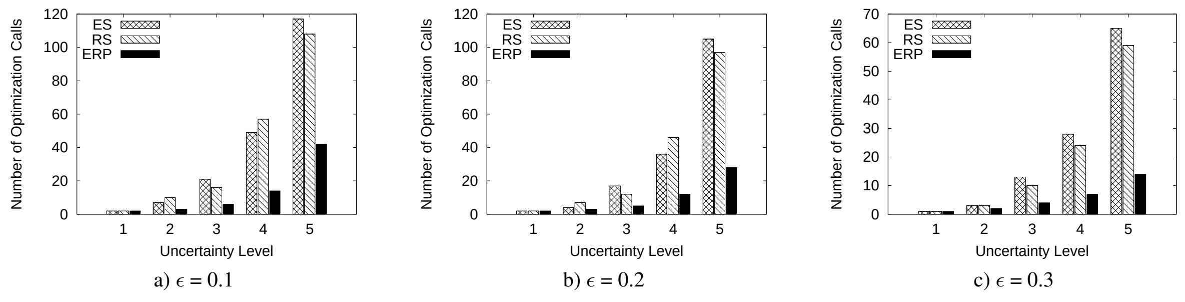
The first class studies the effectiveness of our ERP algorithm for **robust logical plan generation**. Specifically, we compare the *compile-time optimization performance* and the *quality* of the resulting robust logical plans for three alternative techniques. As baseline for the best quality robust logical solution, we employ exhaustive search (ES) over the discretized parameter space. We also implement a search algorithm which randomly selects points in the parameter space as plan candidates (RS). RS stops making optimizer calls if it fails to find a distinct robust logical plan after a given number of optimizer calls. This corresponds to our partitioning technique assigning equal weights to all points in the parameter space. Finally, our weight assignment strategy with early termination is denoted as ERP.

The second class evaluates the effectiveness of our algorithms, $GreedyPhy$ and $OptPrune$, for **robust physical plan generation**. Specifically, we compare different approaches for physical plan generation with respect to their *optimization time* to find the solution and the *space coverage* of the solution. We also measure the total weight of the area covered by the resulting physical plan. As baseline, we again choose the results from the exhaustive search over all load distribution plans, which is guaranteed to find the optimal solution.

The third class is a comparative study assessing the **runtime execution** of the overall RLD system. Specifically, we evaluate the average tuple processing time and runtime overhead of our RLD solution and compare it to state-of-the-art approaches, namely, the resilient load distribution [9] (ROD) and dynamic load redistribution (DYN) [2].

6.3 Effectiveness of Logical Plan Generation

Varying Robustness Thresholds and Uncertainty Levels. This experiment assesses the impact of the robustness threshold ϵ and uncertainty level U on the effectiveness of our **logical plan finder**. The value ϵ of the robustness threshold is varied from 10% to 30% for $Q1$ (5-way join query). Figure 10 shows the number of optimization calls made by ERP, RS and ES. As expected, a lower value for ϵ (tighter robustness bound) results in a higher number of optimization calls, because returned plans cannot be much worse than the corresponding optimal logical plans due to the tight robustness bound ϵ. Furthermore, Figure 10 depicts the optimization efficiency under different uncertainty levels. The higher uncertainty levels result in a larger parameter space. Hence, the number of optimizer calls increases along with the increasing uncertainty level.



a) $\epsilon = 0.1$ b) $\epsilon = 0.2$ c) $\epsilon = 0.3$

Figure 10. Vary Robustness Thresholds and Uncertainty Levels for 3 Partitioning Algorithms

Figure 11 shows the parameter space S coverage of the robust logical plans identified by ES, RS and ERP, respectively. Given the same number of optimization calls, ERP finds more distinct logical plans and thus covers more space than ES. We also observe that ERP is able to completely cover the space given a larger margin on ϵ. This implies that there are many logical plans with trivial cost differences, which agrees with the findings in [12].

In contrast, RS misses many robust logical plans even with the same ϵ as used in ERP. Moreover, ERP provides better parameter space coverage, in fact close to ES in all cases, compared to RS.

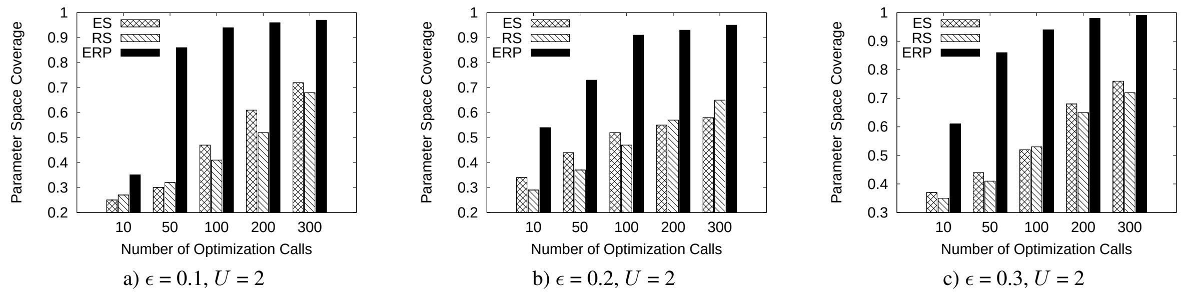


Figure 11. Parameter Space Coverage for 3 Partitioning Algorithms

Varying the Number of Dimensions. Our previous results are based on a fixed number of statistic estimates (i.e., dimensions). We now examine the relative efficiency of the algorithms for dimensions varying from one to ten. $Q2$ (10-way join query) is used because it has a higher number of logical plans compared to $Q1$. Thus, it is more likely to suffer from the exponential growth of complexity with a linear increase in the number of dimensions.

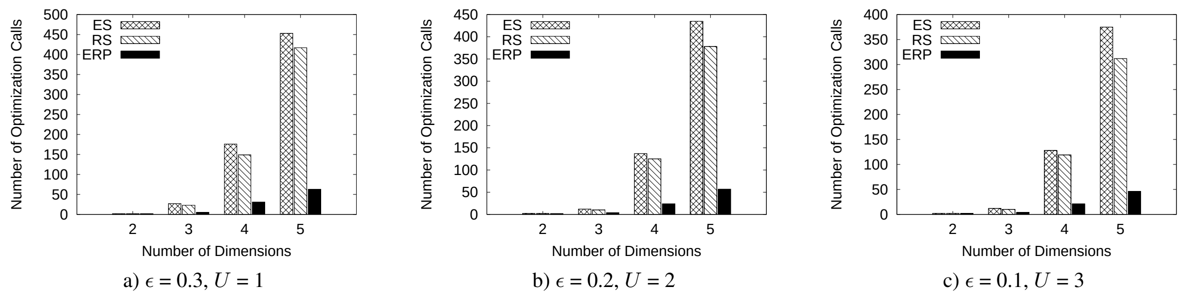


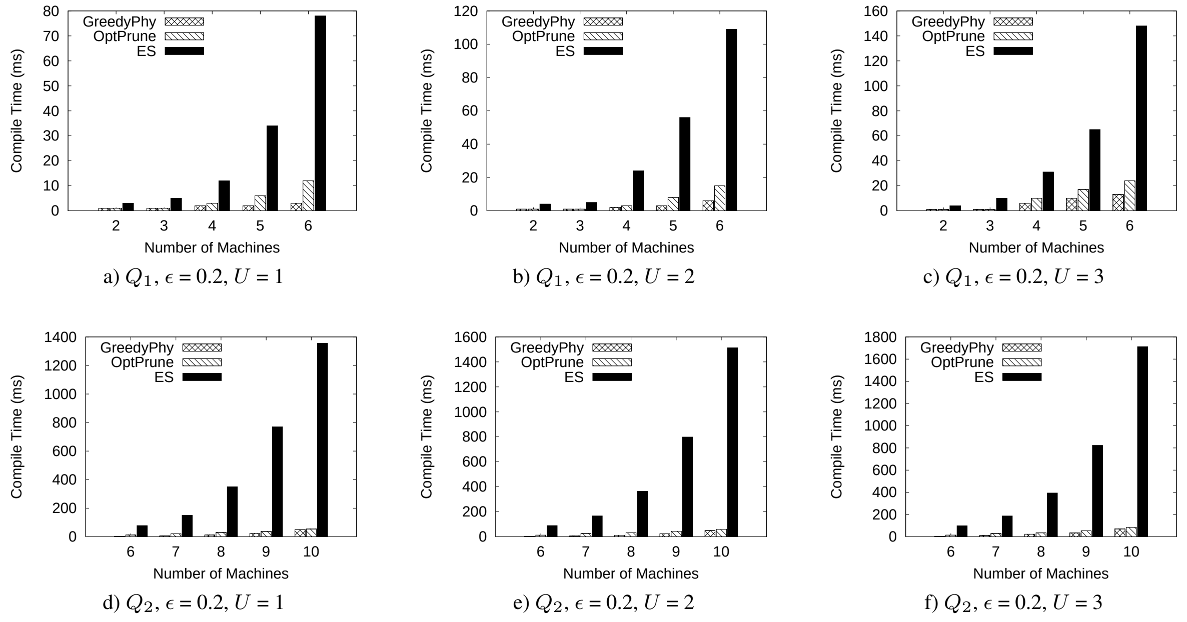
Figure 12. Number of Optimizer Calls for 3 Partitioning Algorithms

We consider 3 parameter configurations to evaluate the efficiency of our algorithm for finding logical plans across diverse scenarios. Our optimizer, as shown in Figure 12, is significantly more efficient than the alternative approaches. It is clear that the more dimensions the parameter space has, the more subspaces are produced by each partitioning step. That is, the number of subspaces grows exponentially with the dimensionality of the parameter space. This issue drastically affects ES because this approach has to check all unit subspaces in the discretized space. As depicted in Figure 12, the number of partitioning iterations increases exponentially with the number of dimensions. In contrast, our proposed ERP solution increases almost linearly by wisely choosing the partitioning areas. Furthermore, we benefit from our early termination mechanism that successfully avoids wasting computations on already robust areas.

6.4 Effectiveness of Physical Plan Generation

Next, we address the relative effectiveness of $GreedyPhy$ versus $OptPrune$ for **physical plan generation**. We measure the quality (i.e., space coverage and associated weight) of the respective algorithms. We vary the number of machines and also use different queries (equi-join of 10 to 20 streams).

Figure 13 shows the average optimization time used by each algorithm for different numbers of operators. $GreedyPhy$ is 12 times faster than its alternatives on average. Exhaustive search (ES) fares the worst because it treats the regions equally in the parameter space. This is a bad choice because ES wastes computations on the margin areas (i.e., less likely to actually occur at runtime) of the parameter space that are less likely to be supported by the resulting physical plan. $OptPrune$ does fairly well compared to ES, because it tends to support the most important logical plans first. Moreover, it uses the result from $GreedyPhy$ as bound for effective branch and bound search. In fact, it exhibits an optimization time similar to our $GreedyPhy$ approach in most cases.



a) Q_1, $\epsilon = 0.2$, $U = 1$ b) Q_1, $\epsilon = 0.2$, $U = 2$ c) Q_1, $\epsilon = 0.2$, $U = 3$

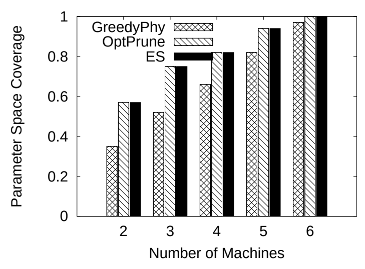
d) Q_2, $\epsilon = 0.2$, $U = 1$ e) Q_2, $\epsilon = 0.2$, $U = 2$ f) Q_2, $\epsilon = 0.2$, $U = 3$

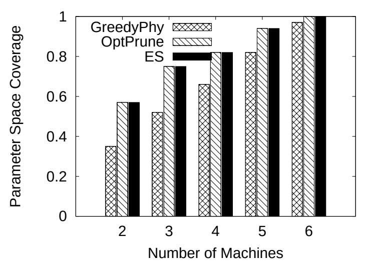
Figure 13. Optimizer Performance for Finding Physical Plan

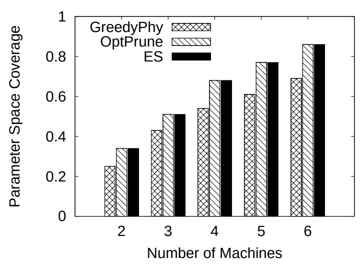
Figure 14 compares the physical plans produced by our $GreedyPhy$ and $OptPrune$ with the optimal solution (ES) for different queries (number of operators) given different system resources (number of machines). We define the average parameter coverage ratio rt_A as a metric to assess the relative effectiveness of the algorithm. The metric is defined as a ratio between the area ($Area(p.phy_A)$) covered by algorithm A's physical plan and the area ($Area(p.phy_{ES})$) covered by the optimal physical plan. The physical plan generated by $OptPrune$ is identical to ES even in the worst scenario, yet the search costs are much cheaper than those of ES. As for $rt_{GreedyPhy}$, the maximum rt is 0.94 and the minimum rt is 0.62 (ratio varies with different queries). Clearly, $GreedyPhy$ sacrifices the quality of the robust physical plan for a reduction in compilation overhead.

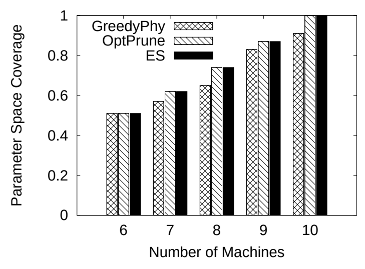
6.5 Measuring Runtime Performance

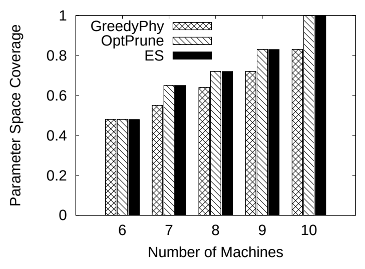
While our overarching goal is to achieve robust processing without load redistribution, DSPSs must process continuous queries in real-time. Thus, we now evaluate the runtime performance (i.e., the average tuple processing time and the total number of tuples produced) of the RLD solution compared to the most prominent approaches, namely, resilient operator distribution [9] (ROD) and dynamic load distribution (DYN). Each query is run for 30 minutes five times using these different solutions with the initial setup for input rates as shown in Table 2. Using synthetic data, we vary stream rates by scaling the rate by a constant. The input data arrival follows Poisson distribution and the input data value follows Uniform distribution described in Table 2. In general, the results in Figure 15 show that RLD outperforms ROD and DYN in both metrics. The primary reason is that neither ROD nor DYN guarantees any optimality of logical query plans since its load migration only changes the operators' physical layout on computing nodes. Consequently, the query processing still suffers from sub-optimal plan


a) Q_1, $\epsilon = 0.2$, $U = 1$


b) Q_1, $\epsilon = 0.2$, $U = 2$


c) Q_1, $\epsilon = 0.2$, $U = 3$


d) Q_2, $\epsilon = 0.2$, $U = 1$


e) Q_2, $\epsilon = 0.2$, $U = 2$

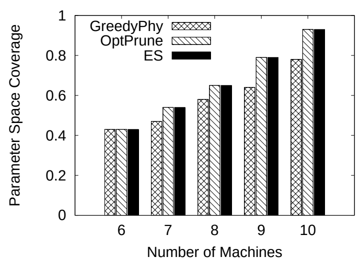
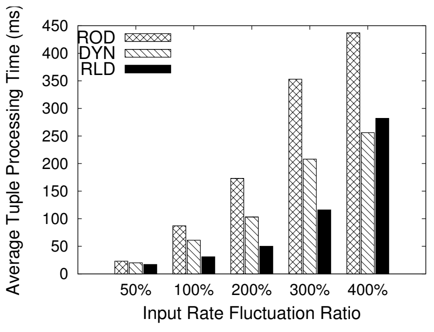

f) Q_2, $\epsilon = 0.2$, $U = 3$

Figure 14. Space Coverage of Physical Plan Generation

execution (ordering) even after the load migration.


a) Average Tuple Processing Time

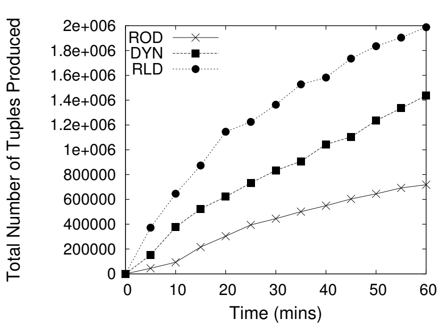

b) Total Number of Tuples Produced

Figure 15. RLD Runtime Performance 1

Average tuple processing time: Figure 15(a) shows the average tuple processing time results over 30 minutes for the algorithms when the input rates vary from 50% to 400% of the initial rates as shown in Table 2. The results in Figure 15(a) over such wide range of fluctuation not only show that RLD is robust to the input rate variations in most cases, but also point out where it fails. When the input rate is low (50%), ROD and DYN are almost as good as our RLD approach. This is because when each operator has only a small amount of load and sufficient machines are available, then no operator migration or switch of logical plans is needed by DYN or RLD, respectively. When the input rate is normal (100%), neither ROD nor DYN's performance does not scale well with the input rate. Our RLD approach performs 3 and 2 times better than ROD and DYN approaches. This is likely due to ROD's performance suffering from executing sub-optimal logical query plans once input data fluctuations arise. DYN approach is also slower since moving operators may result in temporary poor performance due to the execution suspension of those operators. When the input rate is high (200%), the limitation of ROD and DYN become more pronounced. In ROD, a few nodes become bottlenecks. Without load migration, the tuple processing becomes delayed. The load migration in DYN is a passive approach towards tolerating data fluctuations. However, DYN's performance exceeds our RLD approach when the input rate fluctuation ratio is very large (400%). In this case, the load of the system cannot be well balanced since RLD only adopt one physical plan. DYN, on the other hand, performs best with such dramatic fluctuations. The reason is that the computational resources are not sufficient for our RLD approach to handle such fluctuations with one single physical plan. Thus, our RLD approach targets to support fluctuations being known a priori.

Our experiments further inspect whether RLD approach is robust to another two simulation parameters, namely, the number of nodes and the input stream fluctuation periods. Specifically, the input stream fluctuation period is simulated by alternating the input rate of each input stream periodically between a high rate and a low rate. The duration of the high rate interval (i.e., the period of the input stream fluctuation) equals the duration of the low rate interval.

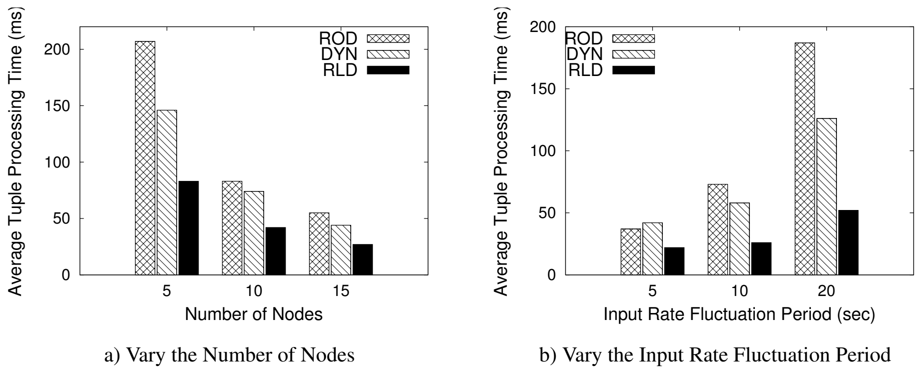


Figure 16. RLD Runtime Performance 2

As indicated in Figure 16, both ROD and DYN do not exhibit the same robustness to the other two parameters as RLD does. In particular, they do not perform well especially when the number of nodes is small (Figure 16(a)), or the load fluctuation period is long (Figure 16(b)). Specifically, when the number of machines is large, the performance difference among all three approaches is small. This is because when each machine has only a small amount of load, all 3 systems succeed to produce stable solutions that require few or no operator migrations. However, our RLD approach still performs better than the other two because it pro-actively switches logical plans at runtime. Thus it can naturally reduce the overall processing load when the statistic changes by executing the most efficient plan ordering at all times.

In terms of the effect of the load fluctuation period, the average tuple processing time of RLD slightly increases while ROD and DYN suffer from the long fluctuation period. The reason is that ROD neither supports load migration nor has multiple logical plans corresponding to different load fluctuations. On the other hand, DYN approach supports load migration, however, its performance is also slow due to the execution suspension caused by operator movements. Our RLD approach avoids load migration and smooths out the fluctuations by exploiting a combined solution of multiple robust logical plans and a corresponding robust physical plan.

Total number of tuples produced: Figure 15(b) indicates the total number of tuples produced by the three load distribution stream models (i.e., ROD, DYN, and our RLD) over 60 minutes. The results demonstrate the sensitivity of three approaches to input stream fluctuations. We increase the input rates from 50% to 100% after the first 20 minutes and further increased the rates to 200% after running for 40 minutes. As indicated in Figure 15(b), our RLD quickly adjusts to the new input rates and continues producing results with the appropriate robust logical plan. On the contrary, ROD barely produces any result tuples after 40 minutes since its physical plan supports a narrower range of data fluctuations without having multiple robust logical plans at its availability. While DYN still can keep up with the new input rates, it again suffers from two factors, namely, the load migration overhead and sub-optimal logical plan execution, respectively. Thus, our RLD approach still performs better than DYN because

RLD's robust logical plans can pro-actively smooth out the load changes.

Runtime Overhead: Now we compare the runtime overhead of RLD and DYN solutions. ROD is not included as it employs a single logical and physical plan and thus incurs no runtime overhead beyond query processing. For both RLD and DYN, we consider any execution costs beyond the actual query processing to be runtime overhead. In RLD, tuples are grouped together into batches and assigned the appropriate logical plan based on the runtime statistics. Thus all tuples in a batch share the same plan. The only runtime overhead incurred in RLD is the initial classification to determine the execution plan for any arriving data, which was measured to be small, on average, about 2% of the query execution costs. On the contrary, DYN suffers from continuous load redistribution overhead. In DYN, the system collects operator statistics and determines which operator to move off the overloaded machine. Multiple factors contribute to DYN's runtime overhead, including the frequency of operator relocation, the state sizes of the moving operators, and the scale of operator relocation. The continuous re-optimization costs of DYN offset the performance gains achieved from using a better physical plan for short-term data fluctuations. In RLD, such overheads are avoided by exploiting a robust physical plan to support multiple logical plans.

7 Related Work

Robust [10, 23] and parametric query optimization [12, 13, 14, 15, 24] are closely related areas. Robust query optimization [10, 23] aims to find one robust query plan that performs reasonably well for known uncertainties in statistics. However, if significant discrepancies exist between estimated and actual values, a single robust plan may fail to prevent performance degradation. Our work instead is unique in that it deploys multiple robust logical plans that together assure coverage across the entire parameter space, which a single plan would not be able to accomplish.

Similarly, parametric query optimization finds a set of plans that are optimal for different parameter settings. The early work [15, 24] optimized a parametric query for *all possible* values of uncertain variables, but postponed the final plan decisions to runtime once the actual statistics become known. Recent works [13, 12] proposes the concept of a plan diagram, a pictorial enumeration of the query plans over the selectivity space. The authors propose to reduce the plan diagram for a query by merging plans whose costs are "close enough" with each other. PPQO [14] tackles the same problem in a progressive fashion. They construct the parametric space and approximate optimality regions.

Our problem faces different challenges compared with the above works [12, 14]. First, in our case the original plan diagram is not given. In fact, it would be extremely expensive to compute such diagram. Instead, we compute robust logical plans based on prediction rather than observation on their cost behavior. Thus, the compile-time overhead is significantly reduced by our solution by not making traditional optimization calls repeatedly. Second, none of the above algorithms are directly applicable to our problem since assumptions made in these works do not consider the physical plan generation with resource limitations in distributed environments. Finally, unlike traditional parametric query optimization, there may be certain regions in the parameter space that cannot be covered by the physical plan produced by our solution due to resource limitations. Our technique uses a probability model to capture the occurrence of points in the space at runtime, and strives to cover the most important regions in that space.

Our work is done in the context of distributed stream processing. The performance and scalability issues in centralized stream processing systems [25, 26, 27, 28] drew attention to distributed stream processing systems [1, 2]. Flux [1] offers dynamic "intra-operator" load balancing by partitioning the input streams into sub-streams and determining the assignment of the sub-streams to servers on the fly. Our work is orthogonal to Flux as we focus on the "inter-operator" load distribution problem. Dynamic load distribution in Borealis [2] minimizes the load variances and maximizes the correlations across all node pairs by *dynamically* distributing loads at runtime. Our work instead produces a physical plan that supports pre-computed (*compile time*) robust logical plans, each designed for a particular region of the parameter space. Thus, it does not rely on runtime load redistribution and avoids costly migration overheads.

Some works have also proposed dynamic load distribution solutions for distributed stream processing [29, 30]. SQPR [29] models query admission, allocation and reuse as a single constrained optimization formalization. Due to the complexity of the problem, it then solves an approximate version. SQPR allocates resources to new queries to be added to a distributed stream system by exploiting the reuse opportunities between new and existing queries to share operator executions. SODA [30] is a stream-based distributed scheduler that optimizes two key metrics, importance and resource utilization, as it makes its scheduling decisions. SODA balances the load across all resources in the system by minimizing a weighted average of metrics that model resource utilization. In addition, SODA controls job admission by weighing how many resources to give to admitted jobs.

Both techniques are related to our work in that both works focus on the physical plan allocation of operators across machines. This layout aspect is covered by both their solutions and our RLD solution. However, their methodologies consider dynamic operator movements across machines as a reactive strategy when an imbalance arises, while our RLD approach does not consider this. Instead, our RLD aims to pro-actively produce a robust load distribution solution $without$ runtime operator migration. Unlike their effort, given known data fluctuations as input, we pro-actively switch among appropriate multiple logical plans at runtime, all of which are executed on the same physical operator allocation.

In essence, both approaches address issues different from our work. Namely, they both focus on (a) query addition (allow queries to be added at run-time), (b) query admission (evaluate expected resource needs of new queries and potentially restrict their execution), and (c) query migration (move operators across machines at run-time). Neither of these three topics are the focus of our work. Moreover, neither method has the explicit goal to produce robust logical plans under known statistic fluctuation ranges. Instead, both of them work with standard single-point estimates.

ROD [9], which we compared against in our experimental study (Section 6.5), produces a load distribution plan to keep the system feasible under workload fluctuations without load migration. However, our work has three key differences from ROD: 1. ROD only focuses on producing a feasible physical plan without exploiting multiple logical plans for a given query. Our work combines principles from parametric query processing with load distribution to obtain a *many-to-one* mapping from a set of robust logical plans to a single physical plan that provides robust query processing performance. 2. The query processing performance is not guaranteed in ROD as it has no knowledge of the logical plans being executed on top of its feasible load distribution plan. On the contrary, our work uses proactive methodology to choose the best logical plan from our robust logical solution to be executed on a single physical solution. Thus, the query processing performance is further improved. 3. The operator distribution algorithm in ROD assumes that the load of each operator is a linear function of input rates with the operator costs and selectivities being constant. In contrast, our work tackles the uncertainty in both selectivities and input rates and their impact on query processing robustness. Consequently, the linear function assumption made by ROD does not hold in our context.

8 Conclusions

The ability to withstand stream data fluctuations is an important consideration in a distributed stream processing system. We design a scalable solution in which a DSPS may benefit from different logical plans at runtime based on varying characteristics of the system. We model the fluctuations in input stream rates and selectivities as a parameter space model. Our ERP then efficiently produces a robust logical solution that covers the space. Taking the robust logical solution as input, $OptPrune$ produces an optimal robust physical plan that supports the logical solution at runtime without operator relocation. Due to the effective bounding strategy, it succeeds to do so with minimal optimization time.

Our experimental results on real world data show the promise of our RLD solution. The average processing time of our *robust* DSPS is significantly reduced compared to all other state-of-art techniques. Thus, this technique is well-suited to modern DSPSs. In the future we will explore advanced issues related to data correlations across streams and in particular synchronized across-stream fluctuation patterns.

References

[1] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[2] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.

[3] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, et al. Scalable distributed stream processing. In *CIDR*, 2003.

[4] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the 1st NSDI*, pages 15–15, 2004.

[5] TradingMarkets. http://www.tradingmarkets.com/.

[6] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[7] R. Diekman and R. Preis. *Load balancing strategies for distributed memory machines*. 1999.

[8] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32:422–469, 2000.

[9] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786, 2006.

[10] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *SIGMOD*, pages 107–118, 2005.

[11] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, pages 106–117, 1998.

[12] Harish D., Pooja N. Darera, and Jayant R. Haritsa. Identifying robust plans through plan diagram reduction. *Proc. VLDB Endow.*, pages 1124–1140, 2008.

[13] Naveen Reddy and Jayant R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, pages 1228–1239, 2005.

[14] Pedro Bizarro, Nicolas Bruno, and David J. Dewitt. Progressive parametric query optimization. *IEEE TKDE*, 21:582–594, 2009.

[15] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. In *VLDB*, page 132151, 1992.

[16] Rimma V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. Self-tuning query mesh for adaptive multi-route query processing. In *EDBT*, pages 803–814, 2009.

[17] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[18] Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. Variance aware optimization of parameterized queries. In *SIGMOD*, pages 531–542, 2010.

[19] Liping Peng, Yanlei Diao, and Anna Liu. Optimizing probabilistic query processing on continuous uncertain data. *PVLDB*, 4:1169–1180, 2011.

[20] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. Mc-Graw Hill, 1984.

[21] Yannis E. Ioannidis and Stavros Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. Database Syst.*, pages 709–748, 1993.

[22] Timothy M. Sutherland, Bin Liu, Mariana Jbantova, and Elke A. Rundensteiner. D-cape: distributed and self-tuned continuous query processing. CIKM '05, pages 217–218, 2005.

[23] Volker Markl, Vijayshankar Raman, David Simmen, et al. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[24] G. Graefe and K. Ward. Dynamic query evaluation plans. In *SIGMOD*, pages 358–366, 1989.

[25] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.

[26] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, et al. Telegraphcq: continuous dataflow processing. In *SIGMOD*, pages 668–668, 2003.

[27] Daniel J. Abadi, Don Carney, et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, pages 120–139, 2003.

[28] Rajeev Motwani, Jennifer Widom, Arvind Arasu, et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[29] Evangelia Kalyvianaki, Wolfram Wiesemann, Quang Hieu Vu, Daniel Kuhn, and Peter Pietzuch. Sqpr: Stream query planning with reuse. In *ICDE*, pages 840–851, 2011.

[30] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 306–325, 2008.