Robust Distributed Query Processing for Streaming Data

CHUAN LEI and ELKE A. RUNDENSTEINER, Worcester Polytechnic Institute

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 reallocation across machines may make these systems ineffective, potentially resulting in data loss or even system failure. To overcome this problem, we propose a comprehensive solution, called the Robust Load Distribution (RLD) strategy, that is resilient under data fluctuations. RLD provides ϵ-optimal query performance under an expected range of 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 in a parameter space that captures the stream fluctuations. The notions of both robust logical and robust physical plans that work together to proactively handle all ranges of expected fluctuations in parameters are abstracted as overlays of this parameter space. Second, our Early-terminated Robust Partitioning (*ERP*) finds a combination of robust logical plans that together cover the parameter space, while minimizing the number of prohibitively expensive optimizer calls with a *probabilistic bound* on the space coverage. Third, we design a family of algorithms for physical plan generation. Our *GreedyPhy* exploits a probabilistic model to efficiently find a robust physical plan that sustains most frequently used robust logical plans at runtime. Our *CorPhy* algorithm exploits operator correlations for the robust physical plan optimization. The resulting physical plan smooths the workload on each node under all expected fluctuations. Our *OptPrune* algorithm, using *CorPhy* as baseline, is guaranteed to find the optimal physical plan that maximizes the parameter space coverage with a practical increase in optimization time. Lastly, we further expand the capabilities of our proposed RLD framework to also appropriately react under so-called "space drifts", that is, a space drift is a change of the parameter space where the observed runtime statistics deviate from the expected optimization-time statistics. Our RLD solution is capable of adjusting itself to the unexpected yet significant data fluctuations beyond those planned for via covering the parameter space. Our experimental study using stock market and sensor network 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.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Distributed Databases

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Distributed system, stream processing, query optimization

ACM Reference Format:
Chuan Lei and Elke A. Rundensteiner. 2014. Robust distributed query processing for streaming data. ACM Trans. Datab. Syst. 39, 2, Article 17 (May 2014), 45 pages.
DOI: http://dx.doi.org/10.1145/2602138

1. INTRODUCTION

Motivation. Distributed Stream Processing Systems (DSPSs) are designed to execute continuous queries over streams of tuples [Shah et al. 2003; Xing et al. 2005; Cherniack et al. 2003; Balazinska et al. 2004]. Continuous query workloads place a heavy burden on precious system resources from CPU processing cycles, memory, and network

This project is supported by the National Science Foundation under grant IIS-091710 and grant IIS-1018443.
Authors' addresses: C. Lei (corresponding author) and E. A. Rundensteiner, Computer Science Department, Worcester Polytechnic Institute, 100 Institute Rd, Worcester, MA 01609; email: chuanlei@cs.wpi.edu.

DOI: http://dx.doi.org/10.1145/2602138

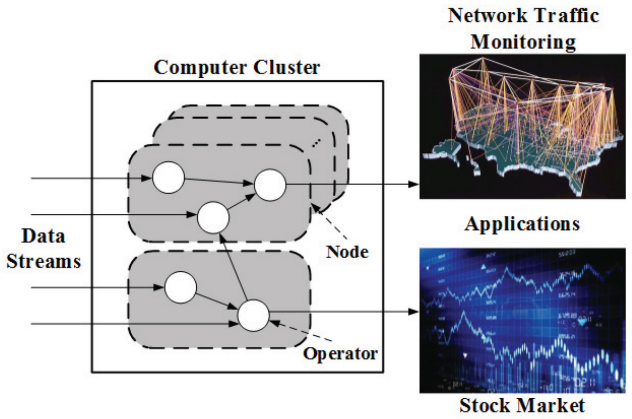


Fig. 1. Distributed stream processing system.

bandwidth. Since workloads can vary greatly, exploiting the limited resources requires effective robust load distribution techniques. Figure 1 depicts a typical distributed stream processing system.

Load distribution, the placement of the operators of a query plan to particular machines (nodes) in a distributed system, is an important design decision impacting the query processing performance [Cherniack et al. 2003]. However, 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 migration of operators across machines may cause delays 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) [TradingMarkets 2013] in recent business news and research.

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

The lookup table *Patterns* contains "bullish" patterns of stock behavior, such as "symmetrical triangle" [TradingMarkets 2013]. Operator op_1 performs a similarity-based join on the latest stock data tuples from the last 60 seconds of the stock stream with the *Patterns* table. Operators op_2 and op_3 match the stock data tuples with the news and research streams on the stock sector and the company name, respectively. Let c_i and δ_i denote the processing costs and current selectivity of op_i respectively.

Suppose it is a bullish market (i.e., stocks are doing well) and the following condition holds: $\delta_1 > \delta_2 > \delta_3$ ($c_1 > c_2 > c_3$). Given these statistics, the best query processing plan (with respect to the number of intermediate results generated) is op_3, op_2, op_1. Assume we have two machines n_1 and n_2 with resources r_1 and r_2 (i.e., CPU, memory, or network bandwidth) available for processing. Then the best load distribution plan with respect to load balancing is op_1 on node n_1, and op_2 and op_3 on node n_2, as depicted in Figure 2(a). Now suppose breaking news reports poor stock performance. This will

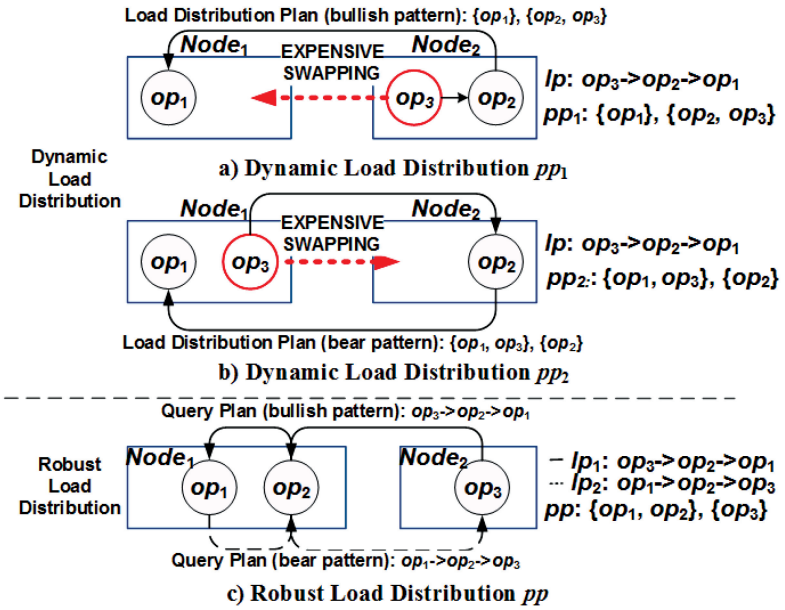


Fig. 2. Dynamic vs. robust load distribution.

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 Distributed Stream Process System (DSPS) has to relocate op_3 to n_1 (see Figure 2(b)). However, in the future if the market exhibits a 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 (see Figure 2(c)), would have supported both query processing plans op_3, op_2, op_1 and op_1, op_2, op_3 from the preceding two scenarios without requiring any workload reoptimization. Clearly, the latter load distribution plan is thus the preferred solution.

In conclusion, two important principles can be derived from the previous example. First, no single logical plan (i.e., query processing plan) may be sufficient for handling all expected variations in statistics. Instead, a set of logical plans must be carefully selected to work together so that under any observed scenario at least one plan among our chosen set of plans will be able to handle this scenario. Henceforth, we call such a set of logical plans a *robust logical solution*.

Second, a *robust physical plan* (i.e., load distribution plan) must support not just one logical plan but instead a set of logical plans, that is, a robust logical solution, to effectively handle all expected fluctuations without repeated reallocations of operators across machines. Thus, we aim to pro-actively design such a dual model composed of a robust logical solution and a robust physical plan that achieves robust query processing performance tolerant to data fluctuations without dynamic load redistribution.

Insufficiency of State-of-The-Art. Traditional distributed and parallel systems [Shirazi et al. 1995; Diekman and Preis 1999] categorize load distribution solutions as either dynamic or static. *Dynamic* load distribution [Shah et al. 2003; Xing et al. 2005] repeatedly improves the current load distribution plan by moving operators across

machines to adapt to load changes. This can come with several drawbacks, including: (1) amortized overhead of repeated load redistributions, (2) missing redistribution opportunities caused by adaptation delays, and (3) expensive maintenance of statistics to undertake appropriate redistributions.

Traditional *static* optimizers [Kossmann 2000] 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. This static approach imposes a relatively low optimization overhead, because its optimization decision is only made once. It cannot, however, 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) [Xing et al. 2006], aims to produce a feasible physical plan that is resilient to time-varying and unpredictable workloads. However, limitations of ROD include: (1) it only focuses on the physical operator distribution without taking logical query plan ordering into consideration, (2) it deals with systems with nonlinear operators by simply transforming their load models into linear ones, which could result in a rather small scope of applicability by ignoring the relationships between the contiguous linear pieces, and (3) it only passively reacts to the changing workloads rather than being proactive to workload fluctuations. Yet, as our experiments confirm, no single logical plan as assumed by ROD can be effective for many different situations at runtime. A detailed discussion comparing RLD to ROD can be found in Section 8.

Our Proposed RLD Approach. In summary, the static approach imposes relatively low optimization overhead, but cannot adapt to changing statistics common in highly dynamic environments. However, the dynamic approach moving operators across machines relatively frequently may result in significant overhead. We thus have developed a practical middle-ground solution between these two extremes called *Robust Load Distribution* (RLD). The main idea of the RLD system is to exploit the key principles from the these load distribution methods while overcoming their respective shortcomings. As foundation of RLD, we introduce the multidimensional parameter-space model, a representation of the uncertainties in statistical estimates of streaming data. We then design the first dual model that pairs a set of *robust logical plans* that exploit optimal plan orderings to support different subspaces of the parameter space with a *robust physical plan* that covers this logical plan set. Such a pair, also called an *RLD solution*, gracefully handles the fluctuations modeled by the multidimensional parameter space. RLD serves as a practical compromise between the two extremes of static versus dynamic optimization.

Contributions. The contributions of this work can be summarized as follows.

(1) We introduce a novel end-to-end RLD solution that overlays a set of robust logical plans with a single physical plan in the parameter space. The RLD solution serves as a practical middle ground between the dynamic and static load distribution approaches. Specifically, we first introduce the notion of ϵ-robust logical query plan that has a cost guaranteed to be within $1 + \epsilon$ bound of the optimal plan within the space. We efficiently compute a multidimensional parameter space representing uncertainties in statistic estimates, including stream input rates and query operator selectivities.
(2) Considering the intractable complexity of the search space composed of all possible subsets of all logical plan combinations, each with their associated physical plan, we adopt a two-step query optimization approach. 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 nontrivial area of the space is shown to be found with high probability. A combination of robust logical plans that together guarantee the performance under any known fluctuation expressed by the parameter space is called a *robust logical solution*.

(3) Given a robust logical solution, we design a family of algorithms for the generation of a robust physical plan, that is, an operator allocation plan tolerant to expected data statistic deviations at runtime without requiring operator reallocation. Specifically, our *GreedyPhy* exploits a probabilistic model to efficiently find a robust physical plan that sustains the most important robust logical plans based on their probability of occurrence at runtime and their respective area of coverage in the parameter space. We further propose the *CorPhy* algorithm to utilize the operator correlation for robust physical plan generation. Such a physical plan is designed to smooth the workload on each node under all expected fluctuations. Lastly, our *OptPrune* algorithm, using *CorPhy* as a bound for search-space pruning, is guaranteed to find the optimal physical plan that maximizes the parameter space coverage at runtime with only an acceptable increase in optimization time.

(4) We further expand the capabilities of our proposed RLD framework to appropriately react when so-called "space drifts" arise. Space drift is a concept drift of the parameter space based on the difference between the expected optimization-time statistics and the observed runtime statistics. We design the *AdaptPhy* strategy that at runtime adapts the current RLD solution, when detecting a space drift, to again become robust under the new defined space. Our RLD solution is thus capable of adjusting itself to the unexpected yet significant data fluctuations beyond those planned.

(5) Our performance evaluation using streaming data from stock market and sensor networks confirms that our RLD solution is superior to state-of-the-art solutions [Xing et al. 2005, 2006] in all aspects, including average tuple processing time, total system throughput, and robustness to data fluctuations. Moreover, our end-to-end RLD approach exhibits a significant improvement on the effectiveness and quality of distributed data stream processing compared to alternative techniques.

Extension from the Conference Version. While an earlier version of this article has appeared in a conference [Lei et al. 2013], the exposition of this manuscript has been updated significantly to not only achieve completeness and improve its readability but also to extend the scope of the proposed work itself.

(1) The earlier conference version only discussed robust physical plan optimization techniques with the assumption of operator independence, while in this article we now establish a correlation-based model to capture the correlation between operators (Section 5.3). This is a natural reflection of the statistical multiplexing effect in DSPSs when multiple operators are packed on the same node. Our newly proposed *CorPhy* algorithm exploits operator correlation knowledge for the physical plan generation to achieve larger parameter-space coverage (i.e., support more robust logical plans). Yet the optimization search is achieved in polynomial time by mapping our problem to a bin packing problem.

(2) Our robust load distribution framework was able to handle data fluctuations within the expected ranges in the earlier conference paper. We now bring our RLD framework one step further by offering a truly robust solution capable of tackling the "space drift" problem, namely the problem that the runtime parameters significantly differ from the expected ones. Our newly proposed *AdaptPhy* algorithm, taking migration costs into consideration, produces a new RLD solution at runtime that is robust in the drifted space using a practical optimization overhead.

(3) We now conduct additional experimental studies to compare the effectiveness of our $CorPhy$ algorithm against our previously proposed algorithms, specifically $GreedyPhy$ and $OptPrune$. In addition, we now also conduct further experiments to demonstrate the effectiveness of our proposed $AdaptPhy$ strategy.
(4) We now elaborate on the execution strategies for multiroute query processing in this manuscript (Section 3). We also present the complete pseudocode for robust physical plan generation for all three strategies that we studied in this work, namely $GreedyPhy$, $CorPhy$, and $OptPrune$.
(5) We include lemmas in Section 2 and 5.4 to clarify our key definitions and principles, and we also develop proofs for all lemmas and theorems. Additionally, we add intuitive examples for each of the proposed major techniques to make the work easy to read and complete.

The rest of this article is organized as follows. We define the RLD problem in Section 2, and overview the overall approach to achieve robustness in Section 3. Sections 4 and 5 describe our algorithms for generating a robust logical solution and the associated robust physical solution, respectively. Handling space drift is discussed in Section 6. The experimental results are presented in Section 7. Sections 8 and 9 discuss related work and the conclusion, respectively.

2. ROBUSTNESS MODEL AND PROBLEM STATEMENT

2.1. Basics of Distributed Query Plans

Common to other distributed stream work [Xing et al. 2005, 2006], we assume the Distributed Stream Processing System (DSPS) is deployed on a shared-nothing homogeneous computing cluster connected by a high-bandwidth network. Hence, network bandwidth is not our key bottleneck. The DSPS accepts a continuous *query* and an expression describing the user's information needs. Then it performs a two-step optimization [Kossmann 2000], namely plan generation and operator placement, to determine the most effective strategy to execute the given query. The plan generation step 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. Multidimensional Parameter Space

Current optimizers [Xing et al. 2005, 2006] 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 multidimensional 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 $< d_1, \ldots, d_n >$, where each d_i is an estimate of the corresponding statistic modeled by that dimension of the space such as selectivity or input rate.

Different methodologies for constructing a parameter space exist, including strict upper and lower bounds [Babu et al. 2005] or levels of uncertainty to the optimizer estimates [Kabra and DeWitt 1998]. We use the latter approach in which the uncertainty level U is computed based on how statistic estimates E are derived. To compute U, we adopted a technique from Kabra and DeWitt [1998] that uses a set of rules to compute uncertainty. Based on the original approach, the value of U characterizes the uncertainty in the estimate E. For example, if a value of E is available from the representative training dataset, then $U = 1$ denotes low uncertainty. For simplicity we henceforth use an integer domain with values from 0 (no uncertainty) to 5 (high

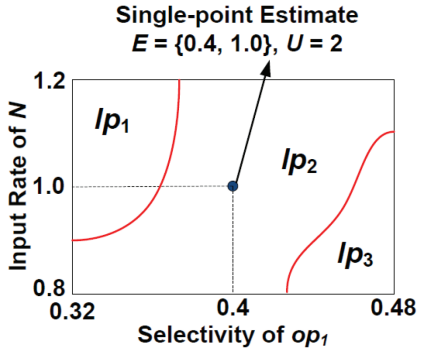


Fig. 3. Selectivity-space example.

uncertainty) to denote the uncertainty levels, though other scales of uncertainty could be easily plugged in, such as stochastic intervals as in Babcock and Chaudhuri [2005]. The most crucial statistic estimates E for query optimization are the selectivities of operators and the input rates of streams [Babu et al. 2005]. In this work, we assume the statistic estimates E and the uncertainty level U correctly represent the data stream fluctuations. If some totally unexpected fluctuation arises in the future that is not within these expected bounds, our current solution may not be able to handle it. In such a case, we designed space-drift handling technology to resolve such scenarios. The parameter space S is computed as shown in Algorithm 1 and Example 2.

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}

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
```

Neither the selectivity for operator op_1 nor the input rate of the News stream may be 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 $E = \{\delta_1 = 0.4, \lambda_N = 100$ tuples/sec$\}$. First, an uncertainty level U (e.g., $U = 2$ median uncertainty) 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.

Similar to parametric queries [D et al. 2008], in practice, each dimension of the parameter space is discretized. For ease of exposition, we henceforth work with a

Table I. Notations and Definitions

Term	Definition
n_i	The ith node in the physical DSPS
OP	All operators in the physical DSPS
op_i	The ith operator in OP
δ_i	The selectivity of op_i
r_i	Resource limit on node n_i
S	Parameter space
pnt	Point 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	All possible logical plans for a given query
LP_j	Robust logical solution for S, a subset of LP
pp	Robust physical plan
$cost(lp, pnt)$	Cost of a logical plan lp at pnt in S
OP_i	a subset of OP allocated to the ith node
$cost(OP_i)$	Cost of a subset of OP on the ith node (i.e., workload on ith node)
lp_{pnt}^{OPT}	Optimal query plan at pnt
Configuration c_i	Association between OP_i and n_i

two-dimensional parameter space, though the extension to higher dimensions is straightforward.

2.3. Notion of Plan Robustness

Let us now consider the most common operators, namely either unary or binary with respect to their inputs, such as select, project, or join. The cost model of a logical plan with two operators can be expressed using the following form that we borrowed from D et al. [2008]

$$cost(lp, pnt) = c_1\sigma_i + c_2\sigma_j + c_3\sigma_i\sigma_j + c_4\sigma_i log\sigma_i + c_5\sigma_j log\sigma_j + c_6\sigma_i\sigma_j log\sigma_i log\sigma_j + c_7, \quad (1)$$

where c_1, c_2, c_3, c_4, c_5, c_6, and c_7 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 these coefficients through standard surface-fitting techniques. To cover all SPJ queries, that is, more operators and all possible combinations of these operators, Eq. (1) can be appropriately extended to model this more complex case. As mentioned in D et al. [2008], such extension is straightforward, with the number of terms in the cost model being $2^{n+1} - 1$.

For example, if op_i and op_j are both selection operators, the cost function of such two operators would be $c_1\sigma_i + c_2\sigma_j + c_7$, while the rest of the terms would be zero. If one of the operators is a projection operation, then c_7 in the preceding equation would have a value to indicate the constant costs of obtaining the projected attributes from the original relation. Regarding the more complex operations such as join, if op_i and op_j share some common attributes, then the cost function would be composed of the term $c_3\sigma_i\sigma_j$ in addition to the terms $c_1\sigma_i$, $c_2\sigma_j$, and c_7. The logarithmic terms $\sigma_i \log \sigma_i$, $\sigma_j \log \sigma_j$, and $\sigma_i\sigma_j \log \sigma_i \log \sigma_j$ apply to operators such as sort and group-by that require multiple passes over the input data. For a more detailed discussion of the logical plan cost modeling, please refer to D et al. [2008].

Given the notations in Table I, we are now ready to introduce the notion of robust query processing. First, we note that Eq. (1) is a monotonic function. Hence, the costs of a query plan can be said to have the property of monotonicity. That is, given a query plan lp and two points $pnt_1 =< d_{1,1}, \ldots, d_{1,n} >$ and $pnt_2 =< d_{2,1}, \ldots, d_{2,n} >$ in

the parameter space S, respectively, and $pnt_1 < pnt_2$, meaning $\forall i$ $(d_{1,i} \leq d_{2,i})$, then we know $cost(lp, pnt_1) < cost(lp, pnt_2)$ for the same plan lp. The following lemma (see Table I for notions) provides a bound on the cost of a logical plan by exploiting the previous monotonicity property. Namely, Lemma 2.1 indicates that the cost of any given plan at any point in the space S must be less expensive than the cost of that plan at pnt_{Hi}.

LEMMA 2.1. *$cost(lp, pnt_i) \leq cost(lp, pnt_{Hi})$, with pnt_i being any point in the space S while pnt_{Hi} is the top-right point in the space S.*

PROOF. For a given parameter space S, the preceding holds true because $pnt_i \leq pnt_{Hi}$ and the plan cost function is monotonically increasing. □

Thus, we can bound the costs of a logical plan for any location within a space S by the optimal logical plan at the top-right corner of S.

Definition 1. Given a parameter space S, a logical plan lp is *ϵ-robust* in S, also called *robust logical plan*, if its costs satisfy the condition (see Table I for notions)

$$cost(lp, pnt_{Hi}) \leq (1 + \epsilon) \times cost\left(lp^{OPT}_{pnt_{Hi}}, pnt_{Hi}\right) \text{ with } \epsilon \text{ between } [0, 1]. \quad (2)$$

From Lemma 2.1, we know that $cost(lp, pnt_i) \leq cost(lp, pnt_{Hi})$ where $pnt_i \leq pnt_{Hi}$ for any plan lp. Also by Definition 1, $cost(lp, pnt_{Hi}) \leq (1 + \epsilon) \times cost(lp^{OPT}_{pnt_{Hi}}, pnt_{Hi})$, if lp is a robust logical plan in the space S. Putting these two inequalities together, we get $cost(lp, pnt_i) \leq cost(lp, pnt_{Hi}) \leq (1 + \epsilon) \times cost(lp^{OPT}_{pnt_{Hi}}, pnt_{Hi})$. In other words, we can assure that the chosen plan lp at any location throughout the space S has a cost guaranteed to be within $1 + \epsilon$ bound of the optimal plan at pnt_{Hi} of S.

Definition 2. The *robust region* of a logical plan lp in S is the largest subarea $S_i \sqsubseteq S$ for which lp satisfies Definition 1 for S_i.

For example, the robust region of lp_1 in Figure 3 covers the left-top corner of the space S.

Definition 3. Given a set of robust logical plans, also called *robust logical solution* LP_j, 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) $cost(OP_i) \leq r_i$, namely for each subset of query operators OP_i assigned to some node n_i, its total cost $cost(OP_i)$ is no greater than the resource capacity r_i of n_i to execute any logical plan $lp \in LP_j$,
(2) $\bigcup OP_i = OP$ means that the union of all OP_i forms the whole operator set OP of the query, and
(3) $OP_i \bigcap OP_k = \emptyset$ $(\forall i, k : 1 \leq i \leq N, i \neq k)$, namely OP_i has no overlap with any other OP_k,

where OP_i denotes the set of operators allocated to node n_i by pp, and OP is the full set of operators in the query. Such physical plan pp (Figure 4 right) is said to *support* a given robust logical solution LP_j (Figure 4 left). Each OP_i associated with n_i is also defined as a *configuration* c_i.

2.4. Problem Statement

Our goal is to identify an end-to-end Robust Load Distribution (RLD) solution that overlays a robust logical solution LP_j with a single physical plan pp in the parameter space S. The key idea is that the resulting system will be able to withstand the known data stream fluctuations, as long as the actual statistics (i.e., input stream rates and

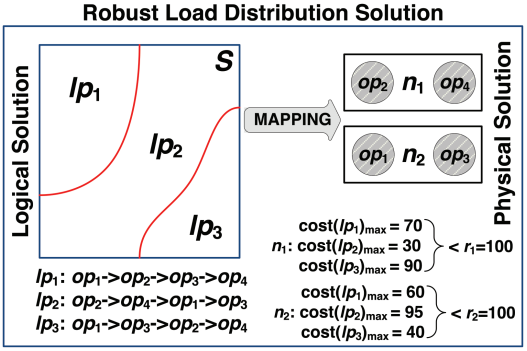


Fig. 4. Robust load distribution solution.

selectivities) remain within the parameter space. Our Robust Load Distribution (RLD) problem can be formalized as follows.

Problem Definition. Given a query q, resources r_k for node n_k ($\forall k : 1 \leq k \leq N$), statistic estimates $E =< e_1, \ldots, e_d >$, and the associated uncertainty levels $U =< u_1, \ldots, u_d >$, the *robust load distribution problem* aims to: (1) identify a robust logical solution LP_j, such that for each point in the parameter space S, there is at least one logical plan lp_i in the solution LP_j that is ϵ-robust by Definition 1 in that subspace S_i, and to (2) produce a physical plan pp that supports the robust logical solution, that is, pp is robust by Definition 3.

Finding the robust solution (Figure 4) for this problem requires a comparison among all possible subsets LP_j of the set of all logical plans LP and all possible physical plans pp in PP in the worst case [Xing et al. 2006]. The number of different logical solutions LP_j is $2^\chi - 1$, where χ is the cardinality of the set of all possible logical plans LP. Moreover, the number of physical plans for a single logical plan is $n^m/n!$ [Xing et al. 2006], where n is the number of nodes and m the number of operators in the logical plan lp. Thus, the search space of the aforesaid problem is a Cartesian product of all possible robust logical solutions LP_j (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 nodes so that this allocation can support a given robust logical solution.

Clearly, the preceding problem is prohibitively expensive. The search space of finding robust logical and of finding physical plans is exponential in the number of query operators and the number of nodes in the system, respectively [Xing et al. 2006]. Furthermore, finding a robust logical solution LP_j and a corresponding physical plan pp supporting LP_j requires the optimizer to search both the logical and physical search spaces. This renders the solution intractable for large problems, such as large numbers of query operators or nodes.

3. OVERVIEW OF RLD APPROACH

Given the intractability of RLD (see problem definition in Section 2.4), we instead employ a two-step approach towards query optimization that is popular for both distributed and parallel database systems [Kossmann 2000] due to its simplification of the overall complexity of the distributed query optimization strategy.

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

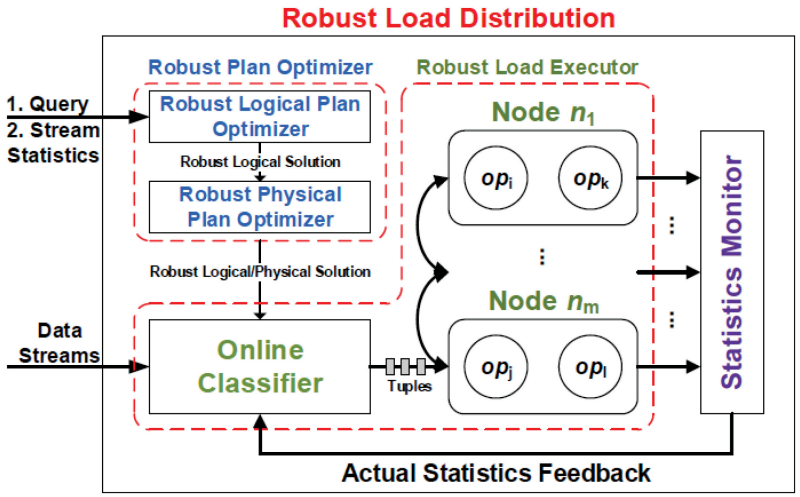


Fig. 5. Robust load distribution architecture.

The preceding two steps efficiently work together to achieve an effective load distribution plan, as our experiments confirm. The first step described in Section 4 has a much smaller optimization complexity than parametric query optimization [Reddy and Haritsa 2005; Bizarro et al. 2009; Ioannidis et al. 1992]. The second step reduces the complexity of producing a robust physical plan by prioritizing robust logical plans to support (Section 5). Our overall RLD strategy conducts load distribution with full awareness of all possible load fluctuations at runtime, thus avoiding costly load redistributions.

We now briefly introduce the key challenges tackled by our RLD solution framework.

(1) How do we efficiently produce a robust logical solution LP_i that maximally covers the parameter space with a probabilistic bound (step 1)?
(2) How can we produce a single robust physical plan *pp* that satisfies all or at least the maximum number of robust logical plans in LP_i by the given resources (step 2)?

The RLD architecture is depicted in Figure 5.

Robust plan optimizer. Our robust plan optimizer tackles the previous two challenges. It 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 exploits a probabilistic model to produce a *robust logical solution*. The robust logical solution guarantees that any missing (i.e., unidentified) robust logical plan cannot occupy a large area in the space. Our robust plan optimizer finally determines a single robust physical plan to support this logical solution by using a family of optimization algorithms. The robust physical plan guarantees that the RLD system avoids load migrations and stays balanced with given resource bounds. The remainder of this article details this optimizer.

Robust load executor. After a robust physical plan is selected, the operators are instantiated by the executor on the machines according to the plan specification. 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. Any technique from the literature on multiple-route query processing [Nehme et al. 2009; Tian and DeWitt 2003] can be plugged in for runtime plan execution.

In our current prototype implementation, RLD is built on top of Query Mesh [Nehme et al. 2009] 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 induced that associates each computed robust logical plan with the specific statistics for which it would be the most appropriate choice for processing the query (see Definition 1). 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 infrastructure, each batch of tuples carries its own execution path (i.e., its own logical plan ordering) stored in a route token (or short r-token). The execution plans in the r-tokens are specified in the form of an operator stack which stores the pointers to the operators' input queues. Each index i corresponds to a unique operator op_i. A batch of tuples is routed next to the operator that is currently the top node in its routing stack. After an operator is done processing the current batch of tuples, the operator pops its index from the top of the routing stack in the r-token. It then puts the processed batch of tuples into the next (now the top) operator's queue. When the r-token operator stack is empty, the batch of tuples is forwarded to the global output queue and thus to the application(s). This is agile, as the system can continuously adapt between logical plans without any plan migration [Zhu et al. 2004].

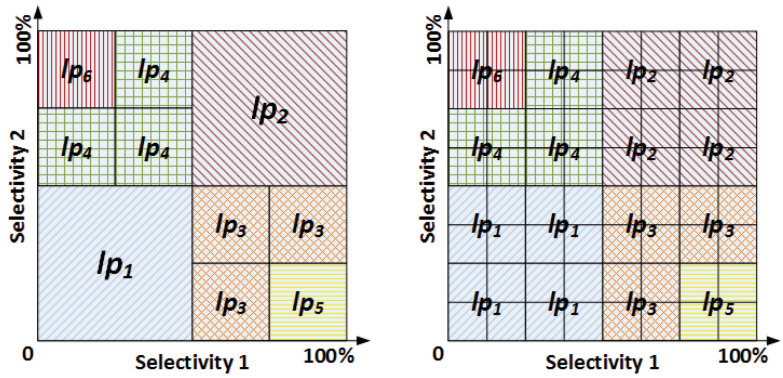
Statistic monitor. The robust load executor requires runtime knowledge about the actual values of key parameters. Thus each machine in a DSPS runs a statistic monitor that periodically samples the selectivity of operators and the associated stream input rates. The monitor then transmits the statistics 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_j such that each plan $lp_i \in LP_j$ satisfies Definition 1. Identifying a robust logical plan requires making an expensive optimizer call, which chooses the most efficient way to execute a query. The exhaustive approach would incur impractically large computational overhead for high-dimensional spaces. Random sampling may also suffer from potentially huge computational costs, but worse yet it tends to fail to identify appropriate robust logical plans in the space, as confirmed by our experiments in Section 7.3. 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 Definition 2). As our experimental study validates, this reduces the space significantly and enables us to find a viable solution in a practical time frame.

Consider a two-dimensional parameter space containing six 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 six plans. The resulting partitioned space will contain 10 subspaces (Figure 6(a)). Specifically, the space will first be partitioned into four subspaces. Logical plans lp_1 and lp_2 are identified as the robust logical plans for their corresponding subspaces. The remaining two subspaces are partitioned again to identify the robust logical plans for each subspace in the bigger subspaces. On the other side, the exhaustive approach depicted in Figure 6(b) would divide the parameter space into a 4×4 grid (making 16 optimizer calls) to discover the same six plans. Worst yet,



(a) ideal approach (b) exhaustive approach

Fig. 6. Comparison of ideal and exhaustive approach.

such overhead increases significantly with the number of dimensions of the parameter space.

Overview of the logical RLD steps. The logical RLD constructor is composed of three critical steps, as sketched next.

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 (Definition 1), we make expensive optimizer calls to verify the robustness of a logical plan in the parameter space. If this tested plan does not satisfy the robustness criteria, a finer-grained partition must be found. Namely, each subspace S_i will be further partitioned if its optimal plan at the bottom-left corner pnt_{Lo} of S_i is not *robust* in S_i (with "robust" defined in Definition 1). 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 our model, we aim to assign higher weights to points where a new robust plan is more likely to exist.

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 generated robust logical plans. In essence, 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 [D et al. 2008; Chaudhuri et al. 2010]. In a nutshell, the latter is about finding all *optimal* plans in a given parameter space, while the former approach instead is about only finding a subset of robust plans, namely those sufficient to achieve good coverage. A detailed comparison of similarities and differences can be found in Section 8, while the efficiency and effectiveness of our proposed strategy are experimentally confirmed in Section 7.3.

4.2. Weight Assignment in Parameter Space

We now describe our strategy of exploiting plan cost information from previously 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 or top-right corners of the associated space. Thus, 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 [D et al. 2008]. 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 Definition 1.

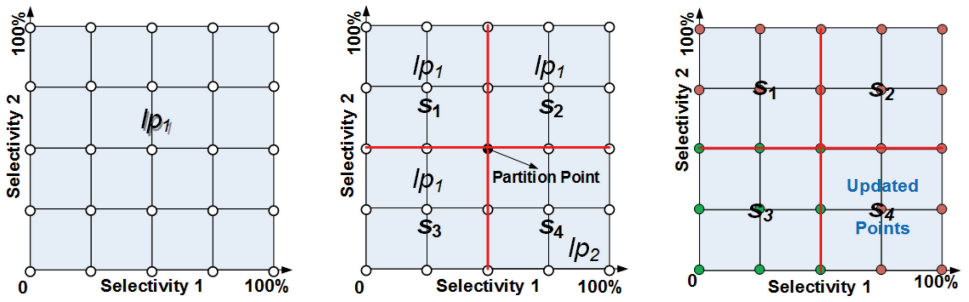
Principle 2. A plan is less likely to be robust at a point if the slope of the plan's cost function, defined in Section 2.3, at that point is high. The slope is the partial derivative of the cost function of the given logical plan with respect to different dimensions. Namely, the slope shows how fast the costs of a logical plan change with each dimension. 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 increases as the slope increases and how quickly the weight decreases as the distance increases. The weight represents how much the slope increases per distance unit, which combines the aforesaid two principles.

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. The weights on each dimension determine the partition point on that dimension. These coordinates together determine the partition point for the parameter space. A point $pnt = (x_1, \ldots, x_n) \in S$ is projected onto each dimension d_i, such that $x_i \in d_i$. A point's weight on each dimension is assigned according to the projected distance between pnt and pnt_{Lo} (see Table I). The weight of pnt in the i-th dimension, denoted by $weight_i(pnt)$, is defined as

$$weight_i(pnt) = min\left(\frac{slope_i\left(lp^{OPT}_{pnt_{Hi}}, pnt\right)}{dist\left(pnt, pnt^i_{Hi}\right)}, \frac{slope_i\left(lp^{OPT}_{pnt_{Lo}}, pnt\right)}{dist\left(pnt, pnt^i_{Lo}\right)}\right). \tag{3}$$

where $dist$ is a function that measures the distance between two points. Any distance measure such as Manhattan or Euclidean Squared Distance [Peng et al. 2011] could be plugged in. When either $dist(pnt, pnt^i_{Hi})$ or $dist(pnt, pnt^i_{Lo})$ is zero, the corresponding ratio between the slope and the distance (e.g., zero) is positive infinity. The min function selects and assigns the other ratio as the weight to pnt. Both $dist(pnt, pnt^i_{Hi})$ and $dist(pnt, pnt^i_{Lo})$ will never be zero at the same time.



(a) original weight (b) select partition point (c) updating weight

Fig. 7. Weight assignment for parameter space.

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

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 previous weight reassignment approach to minimize weight reassignment. That is, we update the weights of points in a subspace as we partition the parameter space S if and only if the following condition is not satisfied.

$$\overline{(lp_{pnt_{Lo}} = lp^{OPT}_{pnt_{Lo}}) \wedge (lp_{pnt_{Hi}} = lp^{OPT}_{pnt_{Hi}})} = False \tag{4}$$

Intuitively, the Eq. (4) condition ensures that the update would not be triggered if the current logical plan at the top-right (bottom-left) corner is identical to the optimal plan identified at the same point after partitioning the space. This guarantees that the current logical plan is robust in the newly generated subspace according to Definition 1.

Example 3. We illustrate the preceding weight reassignment 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 four subspaces, and figure out the optimal logical plans at pnt_{Hi} and pnt_{Lo} for each of the four subspaces S_i (Figure 7(b)). However, we only update the subspaces S_4 (Figure 7(c)), as the just computed optimal plan is different from the predicted robust logical plan at pnt_{Hi} of S_4. Thus, the aforesaid condition reduces the number of subspaces to update by 75% in this case.

Example 3 demonstrates that our proposed method achieves computational savings by avoiding weight reassignments in the area where its current logical plan is identical to the optimal plan identified after partitioning. As stated earlier, the weight reassignment in a subspace only depends on its robust logical plan. If a logical plan is robust in a "large" area, the weights of the points are constant in this area. Thus, our method avoids significant amount of computations as the weights of the points covered by this logical plan remain unchanged after partitioning. The amount of avoided computations grows exponentially with the number of dimensions of the parameter space, as our experiments confirm.

ALGORITHM 2: Weight-driven Robust Partitioning (WRP) Algorithm

Input: A parameter space S, Logical Solution LP_j
Output: A robust logical solution LP_j
1: Plan $lp \leftarrow$ optimize($S.pnt_{Lo}$)
2: **if** lp is *robust* in S **then**
3: $LP_j.add(lp)$
4: **return** LP_j
5: **else**
6: Assign weights to points in S;
7: Choose appropriate point to partition S based on weights
8: **for** $i = 1 \rightarrow n$ (*number of subspaces*) **do**
9: Update weight assignments in $S_i \in S$
10: $WRP(S_i, LP_j)$; // a recursive call consumes a subspace
11: **end for**
12: **end if**
13: **return** LP_j

4.3. Parameter-Space Partitioning

4.3.1. Weight-Driven Robust Partitioning Algorithm. We now illustrate how the previous 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 subspaces, with d denoting the number of dimensions of S. Each subspace S_i will be further partitioned if its optimal plan at pnt_{Lo} is not *robust* in S_i. Finally, the partitioning process stops once the plans at pnt_{Hi} of all S_i are *robust*.

This partitioning process could result in unnecessarily small subspaces 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 subspaces—leading to suboptimal plans. Our experiments (Section 7.3) 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 earlier, *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 subspaces 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 $\epsilon = 0$.

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 Early-terminated Robust Partitioning (ERP), reduces the overhead while providing *a probabilistic guarantee that the robust plans missed cannot occupy a large area in the parameter space*. Our ERP 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 aforesaid 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 the uniform partitioning approach, the probability of finding a new robust plan is proportional to its area of robustness. Given that a finite number of robust plans exist, the probability of finding a new robust plan decreases as we find more robust plans from that constant set of possible plans while 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 predetermined 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 newly generated subspace, if the plan at pnt_{Hi} is a new robust plan that is distinct from all robust plans observed thus far, we reset the aging counter. Otherwise, we increment the aging counter.

Assume that after partitioning t times, we find a new robust plan. Then the aging counter is reset to 0 and we start counting up again. Let Δ 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 $t + \Delta$, since the number of missing robust plans does not change during this interval. We denote the probability of identifying a new robust plan after t partitions within our ERP process by $Pr(t)$,

$$\forall 0 \leq i \leq \Delta - 1, Pr(t + i) = \frac{n_{miss}}{n_{total}} \tag{5}$$

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

THEOREM 4.1. *With a probability of at least $1 - \alpha$, if we do not find a new optimal plan in the partitioning process within Δ trials, where $\Delta \geq \Delta_0 = (1 + \alpha^{-1/2})/\delta$, then the total number of optimal plans not yet found is bounded by $\delta (\leq \delta)$.*

PROOF. Let p be the probability of finding a new robust plan. Then the expected value $\mathrm{E}[\Delta] = 1/p$, and variance $\mathrm{Var}[\Delta] = (1 - p)/p^2$. By Chebyshev's inequality [Papoulis 1984], for any $k \in \mathbb{R}^+$,

$$Pr\left[\left|\Delta - \frac{1}{p}\right| \geq k\sqrt{\frac{1-p}{p^2}}\right] \leq \frac{1}{k^2}. \tag{6}$$

We wish to bound p using Eq. (6). Solving the absolute expression for p gives

$$\Delta^2 - \frac{2\Delta}{p} + \frac{1}{p^2} \geq k^2 \frac{1-p}{p^2}, \tag{7}$$

$$\Delta^2 p^2 + (k^2 - 2\Delta)p + 1 - k^2 \geq 0. \tag{8}$$

Two solutions of the preceding inequality are

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

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

Expressing the bound using p, p_1 and p_2,

$$Pr[p \leq p_1 \vee p \geq p_2] \leq \frac{1}{k^2} \Rightarrow Pr[p \geq p_2] \leq \frac{1}{k^2}. \quad (9)$$

Let $\Delta_0 = (1+k)/\delta$. If $\Delta \geq \Delta_0 = (1+k)/\delta$ then $\delta \geq (1+k)/\Delta$. 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 $\alpha = 1/k^2$ proves the theorem. □

Intuitively, Theorem 4.1 states that if the aging counter reaches a value $\geq \Delta_0$, 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 ERP, which exploits Theorem 4.1 to early terminate the partitioning process, is shown in Algorithm 3.

ALGORITHM 3: Early-terminated Robust Partitioning (ERP) Algorithm

Input: A parameter space S
Output: A robust logical solution LP_j

```
1: LP_i ← φ
2: Stack spaceList ← φ
3: spaceList.push(S)
4: counter_miss ← 0   // aging counter
5: age_threshold ← (1 + α^{-1/2}) / δ   // α and δ defined in Theorem 4.1
6: while counter_miss ≤ age_threshold & !spaceList.isEmpty() do
7:    S_i ← spaceList.pop()   // get the current S_i
8:    p_Hi^OPT ← optimal plan at pnt_Hi of S_i   // do an optimizer call
9:    if p_Hi^OPT ∈ LP_j then
10:      counter_miss++
11:   else
12:      LP_j.add(p_Hi^OPT)
13:      counter_miss = 0   // reset counter_miss since a new robust logical plan is found
14:      weightUpdate(S_i)   // weight assignments in S_i described in Section 4.2
15:      pnt ← maxWeight(S_i)   // choose max weight point in S_i
16:      spaceList.push(S_i.partition(pnt))   // add the list of sub-spaces that compose S_i to
         spaceList
17:   end if
18: end while
19: return LP_j
```

Since the guarantee in Theorem 4.1 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 4.2. *Suppose we stop partitioning according to the aging threshold in Theorem 4.1. If the coverage of an optimal plan lp is area(lp)* $\geq \gamma \cdot \delta$*, for a constant* $0 < \gamma \leq 1/\delta$*, then the probability that the logical plan lp is not found is at most* $e^{-\gamma(1+\alpha^{-1/2})}$.

PROOF. Assume that the total number of robust logical plans is $|LP_j|$ and the total area of the selectivity space is A. Consider the probability of missing a robust logical plan lp with $area(lp) \geq \gamma \cdot \delta A$, denoted $Pr[lp \notin LP_j]$. γ is a constant such that

$0 < \gamma \leq 1/\alpha$. Since $r \geq \Delta_0$ and $1 - \frac{area(lp)}{A} \leq 1$,

$$Pr[lp \notin LP_j] \leq \left(1 - \frac{area(lp)}{A}\right)^{|LP_j|} \leq \left(1 - \frac{area(lp)}{A}\right)^{\Delta_0} \leq (1 - \gamma\delta)^{\Delta_0} \leq e^{-\gamma\delta \cdot \frac{1+\alpha^{-1/2}}{\delta}} = e^{-\gamma(1+\alpha^{-1/2})}. \quad (10)$$

□

Theorem 4.2 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 4.1. From Theorem 4.2 we know that the probability of missing a robust plan decreases exponentially with its area. Thus, while Theorem 4.1 refers to the total uncovered area due to all missing robust logical plans, Theorem 4.2 confirms that individual robust plans with any nontrivial area will be found with high probability.

5. ROBUST PHYSICAL PLAN GENERATION

5.1. Basic Approach for the Generation of Robust Physical Plans

A straightforward strategy for robust physical plan generation would entail the following steps. As input, we accept LP_j, a set of logical plans that together cover the parameter space (see Definition 1) produced by the logical optimizer. Then we compute the set of all possible physical plans for each robust logical plan $lp_i \in LP_j$, denoted by $PP(lp_i)$. Each physical plan $pp_j(lp_i) \in PP(lp_i)$ can support the execution of lp_i within its robustness region without causing any node to become a bottleneck. Thereafter, we find the intersection among all sets $PP(lp_i)$ for all logical plans $lp_i \in LP_j$. If the intersection is not empty, then any physical plan in this intersection is a valid solution that satisfies all robust logical plans in LP_j.

However, if the intersection is empty, then no physical solution exists that supports *all* logical plans in LP_j. We would then need to locate a suboptimal solution. Many strategies are possible. One simple option could be to remove one logical plan lp_i from solution LP_j. Then we repeat the preceding 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 $n^m/n!$ [Xing et al. 2006], 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_j 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.

Therefore, we now propose a family of algorithms that trade off between the optimization complexity and the result optimality.

(1) *GreedyPhy* exploits heuristics to efficiently find a robust physical plan in polynomial time.
(2) *CorPhy* drops the independence assumption which is commonly made in operator placement algorithms, while it finds robust physical plans with larger space coverage compared to *GreedyPhy*, as confirmed by our experiments in Section 7.4.
(3) *OptPrune*, using *CorPhy* as baseline, is guaranteed to find the optimal physical plan to support the maximum number of logical plans, while incurring only a practical 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.

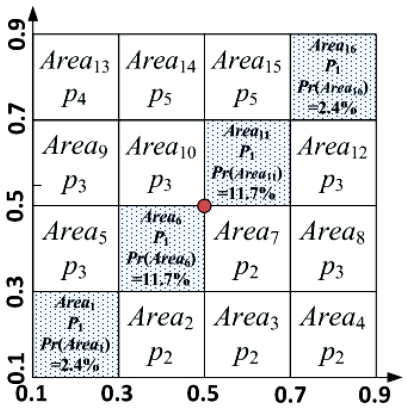


Fig. 8. Weight assignment for logical plans.

(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_j. If not all logical plans can be supported by *pp*, then we drop from LP_j the least important logical plan, namely the plan associated with the *smallest robust region* in the parameter space S from LP_j.

(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 [Peng et al. 2011]. 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 furthest from the given estimated point.

Weight Assignment Policy. Our strategy is to assign a weight to each robust logical plan that incorporates the previous 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 execution time. A robust logical plan's weight, denoted by $weight(lp_i)$, 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 two-dimensional parameter space with each dimension discretized into 16 units (see Figure 8). Suppose the space contains five 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 runtime 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 two-dimensional parameter space, respectively. The preceding assumes that the x and y dimensions are independent following the assumption of independence of selectivity values commonly

ALGORITHM 4: GreedyPhy Algorithm

Input: A robust logical solution LP_j, resources $r_i \in R$
Output: A robust physical plan pp
1: $terminate \leftarrow$ **false**
2: $OP_{max} \leftarrow maxOpCost(LP_j)$ // compute the maximum cost for each operator in LP_j
3: **while** $terminate \neq$ **true do**
4: $pp \leftarrow LargestLoadFirst(OP_{max}, R)$ // try to produce a physical plan for OP_{max}
5: **if** $pp \neq null$ **then**
6: $terminate \leftarrow$ **true**
7: **else**
8: $lp_i \leftarrow getMinWeightPlan(LP_j)$ // find the least important logical plan in LP_j
9: $LP_j.remove(lp_i)$ // remove the least important logical plan from LP_j
10: $OP_{max} \leftarrow maxOpCost(LP_j)$ // update OP_{max} based on the remaining LP_j
11: **end if**
12: **end while**
13: **return** pp

made by query optimizers [Ioannidis and Christodoulakis 1993]. Thus the correlation between dimensions is assumed to be 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 the 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. The *GreedyPhy* algorithm exploits the earlier weight assignment (Algorithm 4). Given a solution LP_j produced by the logical plan optimizer, each plan $lp_i \in LP_j$ has an assigned *weight* $w(lp_i)$. *GreedyPhy* finds the physical plan pp that supports a subset of plans out of the solution set LP_j which are robust to the most frequently occurring data fluctuations captured in the parameter space S.

Given a robust logical solution LP_j and resource constraints $r_i \in R$, *GreedyPhy* calculates a set of all operators OP_{max}, such that the cost of each operator is equal to its maximum cost among all logical plans $lp_i \in LP_j$ (lines 1 and 2). In each iteration the algorithm tries to produce a physical plan by using the Largest-Load-First (LLF) algorithm (similar to the Longest-Processing-Time algorithm [Xing et al. 2006]) (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 found by LLF, it is 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_j (lines 8–10). After repeating lines 3–10 a maximum of $|LP_j|$ times (LP_j is empty after $|LP_j|$ times), the algorithm returns a physical plan pp that supports the most-weighted logical plans selected from LP_j.

Complexity Analysis. The worse case for *GreedyPhy* is that none of the logical plans in LP_j can be supported by the given resources. In other words, it would iterate $|LP_j|$ times before stopping. Therefore, the worse-case complexity of *GreedyPhy* is $O(|LP_j| \cdot |OP|)$ as the complexity of LLF is proved to be $O(|OP|)$ [Xing et al. 2006]. Plus, the complexity of *getMinWeightPlan* is $O(1)$ since all robust logical plans are sorted by their weights in LP_j, and the *maxOpCost* procedure is linear in the number of operators in lp, namely $O(|OP|)$. Therefore, *GreedyPhy* is guaranteed to have a polynomial complexity taking $O(|LP_j| \cdot |OP|)$ time.

5.3. Correlation-Aware Physical Plan Generation

As indicated in Example 1 (see Section 1), the main difference between the dynamic and robust load distribution plan is that the latter approach ensures that the load variation on each node is relatively small. We observe that when the costs of op_1 and op_2 are out of phase, their costs may be naturally balanced such that the total costs of op_1 and op_2 combined actually remain approximately unchanged. Thus, the average cost of each operator is not the only important factor in load distribution. The variation of the cost and the cost correlation between operators are also key factors in determining the performance of a DSPS.

We now introduce another heuristic-driven load distribution algorithm that aims to minimize the cost variance on each node. It is achieved by exploiting the correlation between operators under different scenarios (i.e., robust logical plans). Our strategy drops the load independence assumption which is commonly made in operator placement algorithms [Kossmann 2000; Sutherland et al. 2005; Wolf et al. 2008]. The correlation's intuitive meaning is that when two operators have a positive correlation coefficient, then if the cost of one operator in a certain scenario is relatively large, then the cost of the other operator in the same scenario also tends to be relatively large. On the other hand, if the correlation coefficient is negative, then when the cost of one operator is relatively large, the cost of the other tends to be relatively small. Our algorithm is inspired by the observation that if the correlation coefficient of the costs of two operators is small, then putting these operators together on the same node may help in minimizing the load variance. Namely, the costs of these operator combinations under different robust logical plans would tend to be relatively stable. For example, the total costs of op_1 and op_2 in Example 1 (see Section 1) are relatively stable since when the op_1's costs increase, the costs of op_2 decrease, and vice versa. Thus, the main goal of our correlation-aware strategy is to produce a robust physical plan that sustains different robust logical plans with a minimal load variance on each node.

Given a robust logical solution LP_j (i.e., a set of robust logical plans), the mean and variance of an operator's costs can be captured as

$$\mu_{op_x} = \frac{1}{|LP_j|} \sum_{i=1}^{|LP_j|} cost_i(op_x), \tag{11}$$

$$\sigma^2_{op_x} = \frac{1}{|LP_j|} \sum_{j=1}^{|LP_j|} cost_i(op_x)^2 - \left(\frac{1}{|LP_j|} \sum_{j=1}^{|LP_j|} cost_i(op_x) \right)^2, \tag{12}$$

where $|LP_j|$ is the number of logical plans in the given robust logical solution, and $cost_i(op_x)$ is the cost of the x-th operator in the i-th robust logical plan $lp_i \in LP_j$. Given two operators' cost means and variances, their correlation coefficient ρ and correlation variance σ are defined as follows.

$$\rho_{op_x,op_y} = \frac{\text{cov}\,(op_x, op_y)}{\sigma_{op_x} \cdot \sigma_{op_j y}} \tag{13}$$

$$\sigma_{op_x+op_y} = \sqrt{\sigma^2_{op_x} + \sigma^2_{op_y} + 2\rho_{op_x,op_y}\sigma_{op_x}\sigma_{op_y}} \tag{14}$$

Again, the intuition of utilizing the correlation coefficient between two operators is to find whether or not these two operators mutually balance out their workloads (i.e., small variance of their sum costs). Such correlations and variances represent the relationships of all logical plans in a given robust logical solution. That is, the robust logical solution models all predictable scenarios at runtime. This feature is important

for our strategy as we use these correlations to determine the allocation of the operators to machines (i.e., robust physical plan).

As discussed in Section 5.1, finding a robust physical plan for a single robust logical plan is NP-complete since the problem is equivalent to the bin packing problem. Mapped back to our physical plan generation, each bin represents a physical machine with capacity $r_i \in R$, and an item represents the mean cost μ of an operator. In our case, the operator correlation is further considered in the operator assignments. $CorPhy$ shown in Algorithm 5 succeeds in efficiently producing a robust physical plan which drops the operator independence assumption. Furthermore, the space coverage of $CorPhy$ has a 11/9 OPT + 1 bound (where OPT is the optimal solution from $OptPrune$ described in Section 5.4) from the First-Fit-Decreasing strategy for the deterministic bin packing problem [Chen et al. 2011]. Our experimental results in Section 7.4 confirm the effectiveness of our $CorPhy$ algorithm, especially when strong correlations exist among the operators.

ALGORITHM 5: CorPhy Algorithm

Input: A robust logical solution LP_j, resources $r_i \in R$
Output: A robust physical plan pp

```
1: Initialize lists correlation, correlationVariance and pp ← φ
2: // OP is all operators used in LP_j see Table I
3: for all op_x ∈ OP do
4:    op_x.mean ← computeMean(LP_j, op_x)
5:    op_x.var ← computeVar(LP_j, op_x)
6: end for
7: for all (op_x, op_y) pairs with op_x, op_y ∈ LP_j(x ≠ y) do
8:    correlation_{x,y} ← computeCorrelation(op_x.mean, op_y.mean, op_x.var, op_y.var)
9:    correlationVariance_{x,y} ← computeCorrelationVariance(op_x.var, op_y.var, correlation_{x,y})
10: end for
11: sort OP in decreasing order by op's mean cost
12: while OP ≠ empty do
13:    // return the least loaded machine
14:    m_k ← getLeastLoadNode(R)
15:    // return the operator that has the minimal correlated variance with the operators
       assigned to m_k
16:    op ← getLeastCorOp(m_k, OP)
17:    // add the operator and the associated machine to the physical plan pp
18:    pp.addToPlan(op, m_k)
19:    OP.remove(op)
20: end while
21: return pp
```

The algorithm first initializes empty lists for correlation and variance of each operator's costs (line 1). It figures out the mean and variance of each operator (lines 3–6), and then computes the correlation and variance sum between each pair of operators (lines 7–10). All operators are sorted in decreasing order by their cost mean (line 11). $CorPhy$ then selects the least-loaded machine and finds the operator that has the minimal correlated variance with the operators already assigned to this machine. If a machine already has multiple operators, these operators are considered as one mega-operator and the minimal correlated variance is calculated between this mega-operator and the unassigned operators. In other words, the newly selected operator will have the least impact on its destination node's load variance. This implies that most workload changes of this operator can be smoothed out by the existing operators on that

Table II. Robust Logical Plans and Operator Costs

	op_1	op_2	op_3	op_4	op_5
lp_1	50	60	85	100	40
lp_2	55	40	90	70	60
lp_3	90	90	65	50	70
mean	71.67	63.33	80	73.33	56.67
variance	33.3	25.2	13.2	25.2	15.28

Table III. Correlation and Correlated Variance of Each Pair of Operators

	op_1, op_2	op_2, op_3	op_3, op_4	op_1, op_4	op_1, op_3
correlation	0.89	−0.98	0.68	−0.85	−0.97
correlated variance	56.9	12.54	35.52	17.82	20.75
	op_1, op_5	op_2, op_5	op_3, op_5	op_4, op_5	op_2, op_4
correlation	0.83	0.43	−0.62	−0.99	−0.5
correlated variance	9.42	7.57	6.38	1.26	25.2

machine. This helps the node to tolerate possible runtime fluctuations. The algorithm then assigns the operator *op* to this machine and removes it from the OP list. It repeats the process until the OP list is empty (lines 12–20). The following example illustrates the entire process of the $CorPhy$ algorithm.

Example 6. Given a robust logical solution LP_j and resources R spread over two nodes n_1 and n_2 with resources $r_1 = r_2 = 200$. Assume the robust logical solution LP_j contains three robust logical plans lp_1 ($\langle op_5, op_1, op_2, op_3, op_4\rangle$), lp_2 ($\langle op_2, op_1, op_5, op_4, op_3\rangle$), and lp_3 ($\langle op_4, op_3, op_5, op_2, op_1\rangle$). The associated costs for each operator for each logical plan are shown in Table II. The mean and variance of each operator are included in Table II as well. Consequently, we can compute the correlation and correlated variance for each pair of operators (shown in Table III). According to our $CorPhy$ algorithm, it always returns the least-loaded node and then assigns the operator with least-correlated covariance to that node. In this case, op_3 and op_4 are assigned to nodes n_1 and n_2 respectively since initially nodes $n1$ and $n2$ have zero load and these two operators are the most expensive operators in the list. Now, n_2 becomes the least-loaded node. Then op_1 is selected to be placed on machine n_2 as it has the minimal-correlated variance with the existing operator op_4 on n_2. Next, op_2 is assigned to n_1 as n_1 is the least-loaded node now and op_2 has lower-correlated variance with op_3 compared with op_4. Finally, n_1 and n_2 have almost the same workload, and op_5 is assigned to n_2 as the correlated variance with n_2's existing operator op_1 and op_4 (variance sum $= 9.42 + 1.26 = 10.68$) is less than $n_1(7.57 + 6.38 = 13.95)$. Thus, the robust physical plan is op_2 and op_3 on n_1, and op_1, op_4, and op_5 on n_2. The preceding example illustrates how $CorPhy$ efficiently produces a robust physical plan by exploiting operator correlations.

Complexity Discussion. Our $CorPhy$ algorithm can be seen as a combination of correlation-aware placement and a variant of the Best-Fit-Decreasing strategy [Lodi et al. 2002]. The latter has been shown to have the approximation performance as the First-Fit-Decreasing strategy for the deterministic bin packing problem [Chen et al. 2011]. The best-fit decreasing strategy in $CorPhy$ iterates $|OP|$ times, where $|OP|$ is the number of operators in $|OP|$. The complexity of subprocedures $getLeastLoadNode()$ and $getLeastCorOp()$ are both $O(|OP|)$. Moreover, the complexity of figuring out the mean and variance of each operator, and the correlation and correlated variance between operator pairs are $O(|OP|)$ and $O(|OP|^2)$, respectively. Therefore, $CorPhy$ is guaranteed to have a polynomial complexity of $O(|OP|^2)$ time.

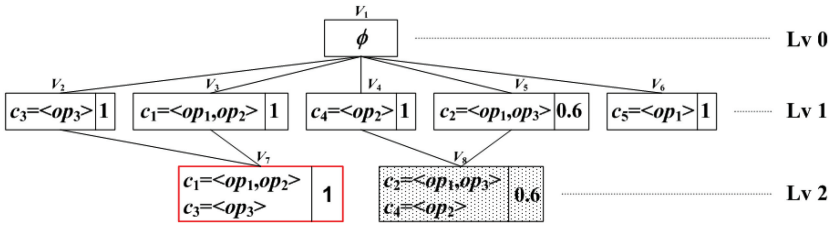


Fig. 9. Robust physical plan search graph.

5.4. OptPrune Physical Plan Generation

The previous algorithms, 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 7).

Search Space. 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 (Figure 9) defined in Definition 4.

Definition 4. A physical plan search graph is a graph $G = (V, E, SCORE)$ where $v_i \in V$ are vertices, $e_{ij} \in E$ are directed edges $v_i \rightarrow v_j$, and a $score \in SCORE$ is associated with each vertex v_i, denoted by $score(v_i)$.

A vertex v_i represents one or more configurations c (see Definition 3). A configuration c is defined as a subset of operators OP_k that together have been assigned to machine m_k. The total costs of OP_k are no greater than the resource capacity r_k of machine m_k to execute any logical plan in a given robust logical solution. For example, if a vertex v_i contains two configurations c_1 and c_2, it means that the partial physical plan has assigned the two subsets of operators (OP_1 and OP_2) to two machines (m_1 and m_2) separately. Configurations in the same vertex v_i must not have overlap with each other. Namely, the intersection of any two configurations in v_i is empty. In other words, each operator in OP is contained in at most one subset in v_i. The *root*, the vertex on level 0, is empty. Each *leaf* corresponds to a *robust physical plan* (i.e., $\bigcup OP_k = OP$). All vertices located between *root* and *leaf* correspond to partial physical plans (i.e., $\bigcup OP_k \subset OP$ but $\bigcup OP_k \neq OP$). A directed edge e_{ij} indicates that the configurations in v_i are a subset of the configurations in v_j. Then the vertex v_j is said to be a *child* of the vertex v_i, and the vertex v_i the *parent* of v_j. The configurations in a child vertex v_j correspond to the union of all configurations in any of v_j's parents. The *score* of a configuration c_k is defined as the sum of all weights (defined in Section 5.2) of any robust logical plans supported by c_k. Intuitively, if a configuration c_k is able to support all logical plans in a robust logical solution, then its score is 1. The *score* of a vertex is determined by the minimal score of the contained configurations in this vertex. Intuitively, if only one configuration in a physical plan cannot support certain robust logical plan, then the entire physical plan cannot support it. Thus, a physical plan is restrained by its configuration with minimal support to the robust logical solution. For example, $score(c_2) = 0.6$ and $score(c_4) = 1$ in Figure 9, then the score of $v_8 = min\{score(c_2), score(c_4)\} = 0.6$.

The key idea is that *OptPrune* can leverage the results generated by *CorPhy* 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, namely $cost(op_1) + cost(op_2) \leq r_i$, where $r_i \in R$ is the machine's resource capacity.

ALGORITHM 6: OptPrune Algorithm

Input: A set of robust logical plans LP_j, resources $r_i \in R$
Output: A robust physical plan pp
1: $pp \leftarrow \phi$ // initialize the physical plan pp
2: $C \leftarrow$ a set of all feasible configurations on a single machine
3: $sort(C)$ // sort by the number of operators in each configuration in descending order
4: Initialize $root$ and $root.children \leftarrow C$
5: $boundingPlan \leftarrow CorPhy(LP_j, R)$
6: $bound \leftarrow score(boundingPlan)$ // the score (see Def. 4) of the physical plan produced by $CorPhy$
7: $pp \leftarrow DFB(boundingPlan, bound, root, pp)$ // call the depth-first search with bound

ALGORITHM 7: Depth First Search with Bound (DFB)

Input: A bounding plan $boundingPlan$, score of boundingPlan $bound$, vertex v, physical plan pp
Output: A robust physical plan pp
1: **if** $v.visited$ = **false** & $v.score > bound$ **then**
2: $v.visited \leftarrow$ **true**
3: **for** each child v_j of v **do**
4: **if** $completeSolution(v_j)$ & $v_j.score > pp.score$ **then**
5: $pp \leftarrow v_j$; $pp.score \leftarrow v_j.score$
6: **else**
7: recursively call $DFB(boundingPlan, bound, v_j, pp)$
8: **end if**
9: **end for**
10: **end if**
11: **return** pp

OptPrune calls its subprocedure *DFB* described in Algorithm 7 to traverse the preceding search graph (defined in Definition 4) in a depth-first search with bounds. *DFB* 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 $pp.score$. *OptPrune* (Algorithm 6) first figures out all possible configurations on a single machine (line 2). The *score* of a physical plan intuitively corresponds to the percentage of the supported area in the parameter space. If a physical plan can support all logical plans in a robust logical solution, this physical plan covers the entire parameter space. Consequently, its score is 1. Before searching the graph, the algorithm sorts the configurations in decreasing order of the number of operators in each configuration and adds these configurations as the children of the *root* of the search graph G (lines 3 and 4). *OptPrune* then figures out its bounding plan and score by calling the *CorPhy* algorithm (lines 5 and 6). *OptPrune* calls its Depth-First search with Bound (DFB) subprocedure by passing in the *CorPhy*'s bounding plan, its score, the *root* of the search graph G, and an empty physical plan (line 7).

The subprocedure *DFB* first checks whether or not the vertex v has been visited before and its score is higher than the bound (line 1). If so, *DFB* sets the vertex v to visited (line 2). Otherwise, *DFB* discards the vertex v and its children in the subsequent search. For example, in Figure 9, if the bound of *CorPhy* is 0.7, then the vertex v_5 and its child v_8 can be safely pruned since v_5's score (0.6) is less than the bound. Each vertex v_j, the child of v, is then checked to verify if it is a complete physical plan and its score is higher than that of the current best physical plan. *DFB* replaces the current best physical plan with v_j and updates its score as well (lines 3–5). If not, *DFB* recursively

calls itself with parameters including the vertex v_j and the current best physical plan (line 7). After returning from recursive calls, *DFB* returns *pp* (line 11).

LEMMA 5.1. *In the directed weighted graph G defined in Definition 4, when we traverse from vertex v_i to one of its children v_j, then score$(v_j) \leq$ score(v_i). After mapping a physical plan search space to this graph G, adding a configuration c_k to the current partial physical plan pp is equivalent to moving from vertex v_i to one of its children v_j. Thus, the score of this newly constructed pp^* is guaranteed to be no greater than that of the original pp, namely, $pp^*.score \leq pp.score$.*

PROOF. Assume a physical plan *pp* contains k configurations and the score of *pp* is *pp.score*. Now let us compare *pp* and its children with $k+1$ configurations including *pp*'s k configurations on the next lower level of the search graph. 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, since the *score* is determined by the minimal weight of the configurations in *pp*. Specifically, if the weight of the newly added configuration is lower than that of the current ones in *pp*, then $pp^*.score$ would be lowered by this newly added configuration. On the other hand, $pp^*.score$ remains the same, even if the newly added configuration has a higher weight than the minimal weight of the current physical plan. Thus the score of each child physical plan cannot be higher than that of its parent *pp*. □

THEOREM 5.2. *OptPrune is guaranteed to find the optimal physical solution (see Definition 4).*

PROOF. If the current *pp.score* is smaller than the bound, then any leaf which has *pp* as its ancestor cannot be the optimal robust physical plan by Lemma 5.1. Thus, the subgraph rooted at *pp* can be safely pruned. The final optimal physical plan can be found by exhaustive search of the rest of the graph. □

Complexity Discussion. The worse case for *OptPrune* is to have to check every physical plan formed by combinations of configurations in the entire search space. Therefore, the worse-case complexity of *OptPrune* is the same as that of exhaustive search, namely, $O(n^m/n!)$, where n is the number of machines and m the number of operators in the logical plan *lp*. 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 7.4). The reasons are twofold. First, *CorPhy* 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. Second, *OptPrune* terminates immediately as soon as 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 that can be supported by the available resources.

6. HANDLING PARAMETER SPACE DRIFT

In dynamic environments, data properties may not only fluctuate but may also undergo large drifts over time. While our RLD approach provides a solution for such a scenario, it does assume that the actual statistics observed remain within the expected parameter space. Even with an initial good choice of an RLD solution, after some time data statistics, for example, data arrival rates and selectivities, may change considerably beyond the expected parameter space, causing a *parameter-space drift* (shown in Figure 10). In this case, the current RLD solution may no longer remain robust. We thus require a new RLD solution to assure that the system continues to remain robust.

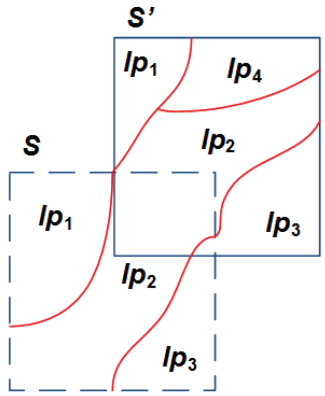


Fig. 10. Selectivity-space example.

In this section, we tackle this challenge of robustness under space drift. Our approach, called Adaptive Robust Load Distribution (ARLD), addresses this space drift problem by abstracting it as a concept drift problem [Mitchell 1997], a well-known subject in machine learning. Such abstraction allows us to discard adaptivity candidates early in the process if they are insignificant or not "worthwhile" to adapt to, and thus to minimize the adaptivity overhead. Our experimental evaluation confirms the effectiveness benefits of our ARLD approach to address space drifts. This completes the last missing piece of robust load distribution.

Our RLD architecture (Section 3) is extensible: new algorithms and modules, including ARLD Statistics Monitor, ARLD Analyzer, ARLD Optimizer, and ARLD Adaptor, can be added without much disturbance to the rest of the system. The ARLD Monitor continuously samples data and execution statistics. Monitored parameters include data values, their frequencies, and the operators' costs and selectivities. Our monitoring approach is comparable to those found in the literature [Babu et al. 2004; Chaudhuri et al. 2004]. Given the measurements from the ARLD Monitor, our ARLD Analyzer then determines if a space drift has occurred. Based on this analysis, the ARLD Optimizer makes recommendations if and how the RLD solution should be adapted. The ARLD Adaptor takes these recommendations and physically adapts the RLD solution from the old to the new plan.

Monitoring, reoptimization, and migration in ARLD add overhead to query processing. Thus, to minimize the overhead, the following design guidelines to drive our solution: (1) monitoring should be lightweight, and only if significant or long-term changes are detected should the more expensive reoptimization process be invoked, (2) the decision to adapt should be made only if significant improvement in the performance is expected from the resulting new robust solution, and (3) the physical migration overhead should be kept low.

6.1. ARLD Monitoring

Monitoring aims to identify if the current statistics are no longer consistent with the current data and its characteristics. ARLD Monitor employs two techniques, namely the input data monitoring and the execution statistics monitoring.

Input Data Monitoring. For data stream monitoring, we sample the arriving data to check if changes in the data values and their distributions have occurred. Monitoring data values (in addition to the distributions and execution statistics) has the advantage that the adaptive system can exploit this extra information to minimize the overhead

of the more expensive execution statistics monitoring. If the system detects a change in data streams (e.g., arrival rate of one of the streams), it may then employ the more expensive execution statistics monitoring to see if the RLD solution may need to be adapted. We adapt simple random sampling [Lipton et al. 1993], which is one of many possible data sampling techniques we could deploy for data stream monitoring.

Execution Statistics Monitoring. The statistics collected during execution track the selectivities and costs of operators. We instrument query operators to collect three types of statistics: (1) independent selectivities, (2) correlated selectivities, and (3) operator costs (measured by wall-clock time). All three types of statistics are collected for each robust logical plan. To compute independent selectivities, a specially marked batch of tuples can be used [Nehme et al. 2009], which continues to travel through all operators without being filtered out. For correlated selectivities, the selectivity is computed simply based on the intermediate tuples that have not been discarded by any previous operators in the execution and thus reach the current operator. All three types of statistics are collected for each individual logical plan by the operators.

6.2. ARLD Analyzer

The ARLD Analyzer takes the data samples and the statistics from the ARLD Monitor and determines if any significant drift of the parameter space has occurred. A drift of the parameter space occurs when data and execution statistics have changed significantly beyond some threshold, implying that the RLD solution may need to be altered. The analysis consists of two phases: (1) parameter-space reconstruction, and (2) space drift detection. If the ARLD Analyzer determines that there is indeed a space drift, then it calls the ARLD Optimizer to make recommendations for a new RLD solution.

Parameter-Space Reconstruction. Given the collected statistics from the ARLD Monitor, the ARLD Analyzer constructs a new parameter space S'. The central point of this new space S' is the average of the collected statistics on each dimension. The range of each dimension is no longer calculated based on the uncertainty level, rather it is based on the actual lowest and highest values periodically observed at runtime.

Space Drift Detection. Since the RLD (re)optimization and migration are expensive, only significant space drift will be reported by the analyzer. The significance requires a distance measure $dist_{space}$ which quantifies the difference between S and S' if $dist_{space} > \theta_{space}$, where θ_{space} is the adjustable distance threshold. The key to drift detection is the intelligent choice of the distance function to compute $dist_{space}$, which must accurately quantify a data change that may impact the current RLD. The choice for the threshold θ_{space} value defines the balance between the sensitivity and the robustness of the detection. The smaller θ_{space}, the more likely we are able to detect small drifts in the data streams, but the larger is the risk of a false positive.

One approach to measuring such data differences is to first estimate the probability distributions of the data, and then compute the distance, such as the *Kullback-Leibler Divergence* or the *Jensen-Shannon Divergence* [Cover and Thomas 1991], between the estimated distributions. However, this approach is computationally impractical for large and high-dimensional data. The problem becomes even more challenging in streaming data environments, as the high speed makes it difficult for such expensive algorithms to keep up with the data [Wang and Pei 2005]. To tackle this issue, we exploit an efficient method for determining the space coverage difference. Using the current RLD, each logical plan may or may not be robust in any subregion in the new space S'. We examine the robust region of each logical plan according to Definitions 1 and 2 in Section 2.3. The total space coverage of S' is a summation of the robust regions of all its logical plans. Thus, we can easily measure the difference between the space

coverage by the current RLD between the old spaces and the new space S'. This way we evaluate space drift by comparing the space coverage. This method is extremely efficient, since all it requires is a quick verification over a fixed number of robust logical plans in the current robust logical solution.

6.3. ARLD Optimizer

ARLD Optimizer is designed to recommend a new RLD solution to the adaptor based on the analysis for future query processing. Given the updated statistics from the monitor, a new RLD solution is computed and its space coverage of the new parameter space S' is determined and compared to the current RLD solution's coverage of S'. If the newly computed RLD' has a bigger coverage, then it is considered as a potential migration candidate. To evaluate the benefit of migration to the new solution and to select the best new RLD solution, ARLD Analyzer uses the metric, called improvement I

$$I(RLD, RLD', S') = 100\% * \frac{S'_{cover} - S' \bigcap S}{cost(migration)}, \tag{15}$$

where RLD is the initial and RLD' the newly recommended solution, S'_{cover} is the expected coverage of S' that a physical plan pp' of RLD' can support, and $S' \bigcap S$ is the overlapping area between the original space S and the drifted space S'. The subtraction informs the analyzer how much extra area in S' that a pp' can support besides the already supported areas in S. The $cost(migration)$ refers to the costs of migrating from pp to pp'. The ARLD Optimizer computes the expected improvement value, and if the value is deemed substantial, then the migration to pp' is recommended. Namely, the new RLD' (i.e., the new robust logical solution + the new robust physical solution) is recommended.

Now, we study how the ARLD Optimizer finds a new RLD solution that could be installed with minimal migration costs. Given the updated statistics, the ARLD Optimizer first forms the new parameter space S'. Then it invokes the logical optimizer (Section 4) to produce a new robust logical solution LP'. With the new robust logical solution, our goal is to adjust the current robust physical plan pp with minimal migration costs so that the new robust physical plan pp' supports LP'. As discussed in Section 5, to search through all possible physical plans is infeasible for a large number of machines or a large number of robust logical plans. Thus, we now propose a cost-based allocation-aware heuristic that finds a good RLD solution in reasonable time without exhaustive enumeration of the search space. Our goal is that the resulting RLD solution should have a better coverage over the space S' than the old solution RLD, and minimal migration overhead of adapting the current physical plan pp to the new robust physical plan pp'. Namely, our optimization goal is to maximize the value of improvement I.

The *AdaptPhy* heuristic has the following steps.

(1) Initially, the starting physical solution is set to be the current robust physical solution, namely pp.
(2) Iteratively, the search algorithm traverses physical plan search space to find another solution pp'.
(3) The coverage of S' is computed and compared to the coverage of pp found so far. If pp has a bigger coverage than pp, then the current physical solution is replaced with pp'. Steps 2 and 3 are repeated until the solution no longer improves in the last several rounds, indicating that the search process has reached a plateau, or the given time bound or resource bound is exhausted.

Search Strategy. We employ a Simulated Annealing (SA)-based approach [Bertsimas and Nohadani 2010] as it is known to locate a good approximation of the global optimum

of a function in a large search space. That is, random movement in the search space is tolerated by initially setting the "temperature" parameter to high. Later less and less random movement is allowed by lowering the "temperature", until the solution settles into a final "frozen" state. This allows the heuristic to sample the solution space widely to jump over local minima when the "temperature" is high, and then gradually move towards simple steepest descent as the "temperature" cools. The search can move out of local optima during the high-temperature phase. The simulated annealing algorithm accepts a worsening move with a certain probability. This probability declines as τ declines, by analogy the randomness in the movements decreases as the temperature falls. When τ is small enough the algorithm accepts only the improving moves. Algorithm 8 sketches the pseudocode for SA physical solution search.

ALGORITHM 8: AdaptPhy Algorithm

Input: Current robust physical solution pp
Output: New robust physical solution pp'
1: $pp \leftarrow currentPhy$
2: Choose an initial (high) temperature τ
3: Choose a value for ρ, the rate of cooling parameter
4: $I \leftarrow 0$
5: $pp' \leftarrow$ a random neighbour of pp
6: $I' \leftarrow \frac{S'_{cover} - S' \bigcap S}{cost(migration)}$
7: $diff \leftarrow I' - I$
8: //Decide to accept the new physical solution pp' or not
9: **if** $diff \geq 0$ **then**
10: $pp \leftarrow pp'$ //pp is better than or same as pp
11: $I' \leftarrow I$
12: **else**
13: $pp \leftarrow pp'$ with probability $e^{\frac{-I}{\tau}}$
14: **end if**
15: **if** stop condition is met **then**
16: **return** pp
17: **else**
18: reduce temperature by setting $\tau = \rho \times \tau$, and go to Line 5
19: **end if**

6.4. ARLD Adaptor

The ARLD Adaptor takes the migration plan from the ARLD Optimizer and physically adapts the RLD solution from the old to the new plan. The movement of states of stateful operators across machines has been studied in the literature [Abadi et al. 2005; Motwani et al. 2003]. While any method could be gracefully plugged into our system, here we use the method proposed in Borealis to move an operator state from one machine to the next at runtime [Abadi et al. 2005].

To accomplish the migration, the execution framework needs to switch to the new logical solution received from the ARLD Optimizer. A single pointer reassignment to the new classifier object (described in Section 3) corresponds to the actual execution of RLD migration. What makes this possible is the architecture of the RLD framework. The physical separation between the component that determines which plans should be used for execution and the component that actually executes the plans based on specifications makes the RLD adaptivity so lightweight. To change the execution strategy, all the system needs to do is to modify the specification of the plan (in the classifier).

Table IV. System Parameters and Distribution Distribution

Parameter	Value	Description
Data Arrival	*Poisson*	Data arrival distribution
μ	500 msec	Mean inter-arrival rate
$\lvert T_{dq} \rvert$	1,000	Maximum # of tuples dequeued by an operator at a time
Ruster size	100 tuples	Minimum *ruster* size
Data Distributions		
Uniform ($\alpha = 0, \beta = 100$): *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 ($\lambda = 1$): *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		

7. EXPERIMENTAL STUDY

We have implemented the proposed techniques on D-CAPE, a distributed continuous query processing infrastructure employing stream query engines [Sutherland et al. 2005] over a cluster of shared-nothing processors. The experiments were run on Linux machines with AMD 2.6 GHz Dual-Core CPUs and 4GB memory.

7.1. Datasets and Queries

Real-life datasets. We have employed two real-life datasets in our experiments. Stocks-News-Blogs-Currency dataset contains NYSE stock prices [Nehme et al. 2009], foreign currency exchange rates from Yahoo! Finance, and news and blogs via RSS feeds. The stock dataset was continuously collected by our data pulling application. The original timestamps were attached to each tuple in the data stream. In our experiments, we serve the data to our stream processing engine based on these timestamps in a push-based fashion. Within a window, the number of tuples at times varies significantly from one to next, which indeed reflects data fluctuations. Sensor dataset contains readings from sensors in the Intel Research, Berkeley Lab [Babu et al. 2005]. The sensor readings are streamed to the distributed stream processing engine in the order they are generated, as if they were submitted by sensors in real time. Default properties and system parameters are depicted in Table IV.

Synthetic datasets. To study the effectiveness of our strategies under data fluctuations, we use well-known distributions: uniform and Poisson. These distributions are selected to model many real-life phenomena [Allen 1968; Hua et al. 1993]. An application example for uniform distribution would be long-term patterns of data. Regarding Poisson distribution, the real-life applications include service times in a system, the number of phone calls at a call center per minute, and the number of times a Web server is accessed per minute. This enables us to control a variety of parameters, such as join selectivity and stream input rate, to assess their impacts. Default properties, system parameters, and data distributions are depicted in Table IV.

Queries. We deploy N-way join queries, as those are among the core and most expensive queries in database applications. The default settings used in our experiments are listed in Table IV. The queries are equi-joins of 10 or 15 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 dataset is streamed into the system or the scheduling of tuple processing (i.e., achieving repeatable workloads).

7.2. 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, which always chooses to partition the space using the central point. The granularity of the discretized parameter space varies. The number of units for each dimension remains the same, and the number has been set to 100 throughout our experimental study. 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 values of ϵ and δ are set to 0.05. This means that if we do not find a new robust logical plan after considering 110 samples, then the sum of areas covered by the missing robust logical plans is less than 0.05 with a probability of at least 0.95.

The second class evaluates the effectiveness of our algorithms, $GreedyPhy$, $CorPhy$, 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, that is, clearly the larger the area covered, the better the robust 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 (ROD) [Xing et al. 2006] and dynamic load redistribution (DYN) [Xing et al. 2005].

In summary, the first two classes demonstrated the efficiency and effectiveness of our robust logical and physical optimization algorithms. These algorithms exploit single-point estimations and uncertainty levels which capture the characteristics of any given datasets. The third class evaluation shows similar trends with both synthetic and real-life data (stock and sensor datasets). Due to lack of space, the majority of results presented in the following sections involve synthetic datasets.

7.3. 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 the query $Q1$. The query $Q1$ is an equi-join of 10 streams. Namely, the query has the form $A \bowtie B \bowtie C \bowtie D \bowtie E \bowtie F \bowtie G \bowtie H \bowtie I \bowtie J$. The selectivity of operators varies due to the data distribution and input rate changes at runtime. Figure 11 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 11 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.

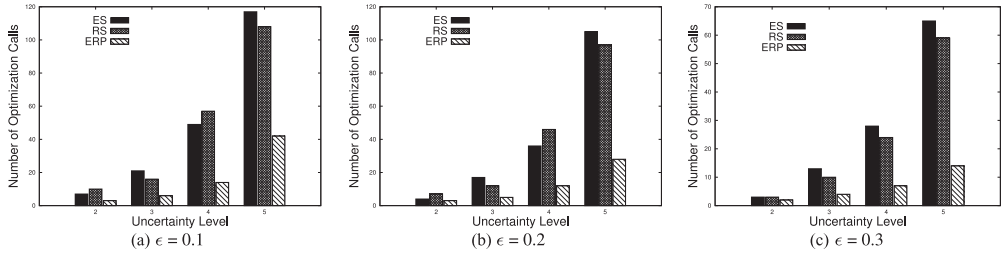


Fig. 11. Optimization efficiency (i.e., number of optimization calls required) when varying robustness thresholds and uncertainty levels for three partitioning algorithms for robust logical solution generation.

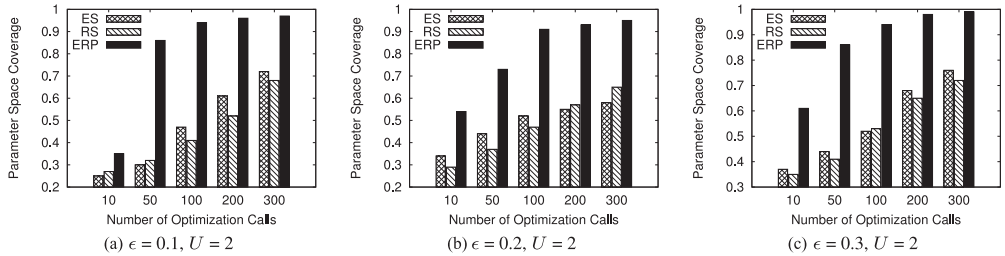
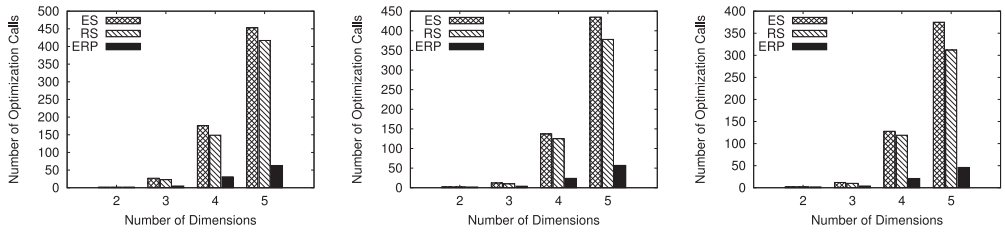


Fig. 12. Parameter-space coverage achieved by three partitioning algorithms for robust logical solution generation when varying number of optimization calls.

Figure 12 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 even when given a larger margin on ϵ. This implies that there are many logical plans with trivial cost differences, which agrees with the findings in D et al. [2008]. In contrast, RS misses many robust logical plans even with the same ϵ as the one used in ERP. Moreover, ERP provides better parameter-space coverage, in fact close to ES in all cases, compared to RS.

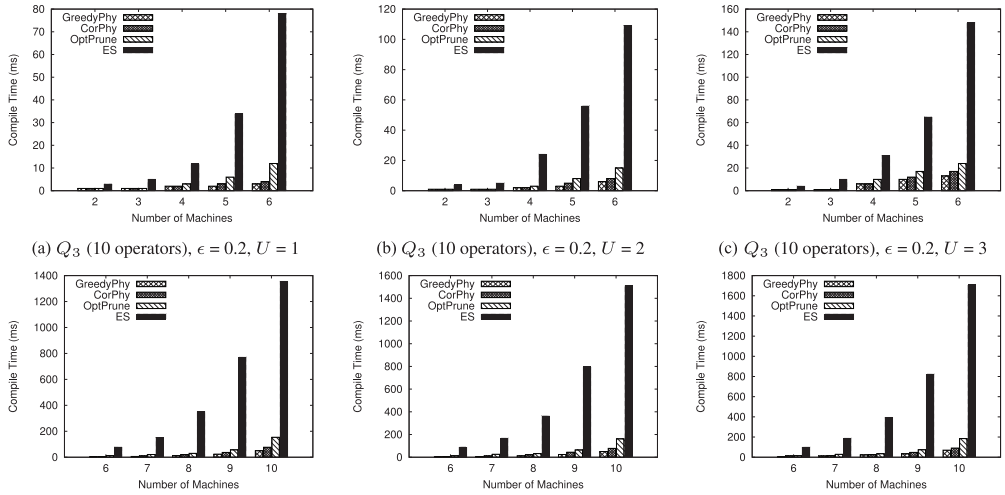
Varying the Number of Dimensions. Our previous results are based on a fixed number of statistic estimates (i.e., dimensions). We now examine the relative efficiency of the algorithms for dimensions varying from one to ten. $Q2$ (15-way join query) is used because it has a higher number of logical plans 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.

We consider three parameter configurations to evaluate the efficiency of our algorithms for finding logical plans. Our optimizer, as shown in Figure 13, 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 13, 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.



(a) $\epsilon = 0.3$, $U = 1$ (b) $\epsilon = 0.2$, $U = 2$ (c) $\epsilon = 0.1$, $U = 3$

Fig. 13. Number of optimizer calls by three partitioning algorithms for robust logical solution generation when varying the dimension of parameter space.



(a) Q_3 (10 operators), $\epsilon = 0.2$, $U = 1$ (b) Q_3 (10 operators), $\epsilon = 0.2$, $U = 2$ (c) Q_3 (10 operators), $\epsilon = 0.2$, $U = 3$

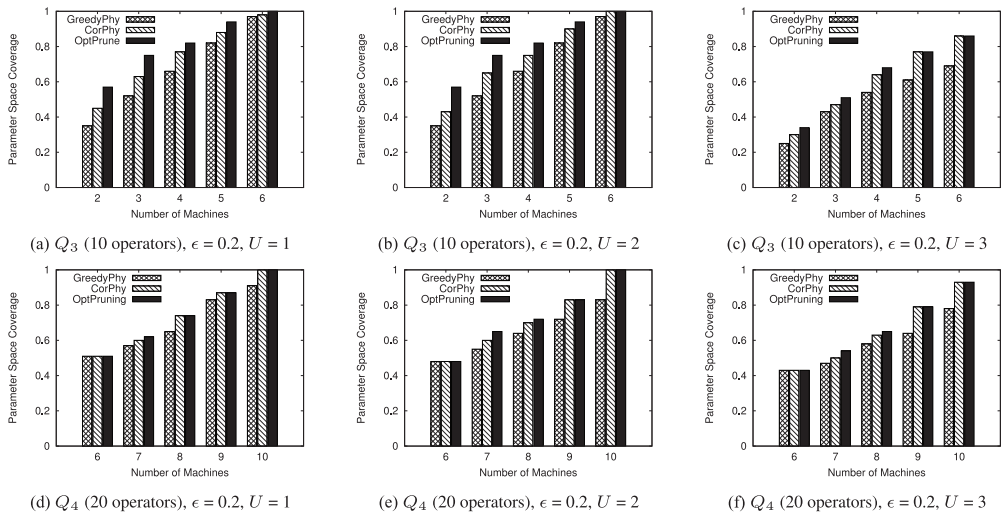
(d) Q_4 (20 operators), $\epsilon = 0.2$, $U = 1$ (e) Q_4 (20 operators), $\epsilon = 0.2$, $U = 2$ (f) Q_4 (20 operators), $\epsilon = 0.2$, $U = 3$

Fig. 14. Optimizer performance for finding robust physical plans when varying queries and uncertainty levels.

7.4. Effectiveness of Physical Plan Generation

Next, we address the relative effectiveness of *GreedyPhy*, *CorPhy*, and *OptPrune* for physical plan generation. We measure both the optimization time as well as 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 14 shows the average optimization time used by each algorithm for the queries with different numbers of operators. *GreedyPhy* and *CorPhy* are one order of magnitude faster than their alternatives on average. As discussed in complexity analysis in Sections 5.2 and 5.3, *GreedyPhy* and *CorPhy* take $O(n)$ and $O(n^2)$ time, respectively. The difference between *GreedyPhy* and *CorPhy* is trivial in Figure 14 since the number of operators in both queries is 20 at most. 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 *CorPhy* as bound for effective branch-and-bound search. In fact, it exhibits an optimization time similar to our *CorPhy* approach when the numbers of operators and machines are



(a) Q_3 (10 operators), $\epsilon = 0.2$, $U = 1$ (b) Q_3 (10 operators), $\epsilon = 0.2$, $U = 2$ (c) Q_3 (10 operators), $\epsilon = 0.2$, $U = 3$

(d) Q_4 (20 operators), $\epsilon = 0.2$, $U = 1$ (e) Q_4 (20 operators), $\epsilon = 0.2$, $U = 2$ (f) Q_4 (20 operators), $\epsilon = 0.2$, $U = 3$

Fig. 15. Parameter-space coverage of physical plan generation.

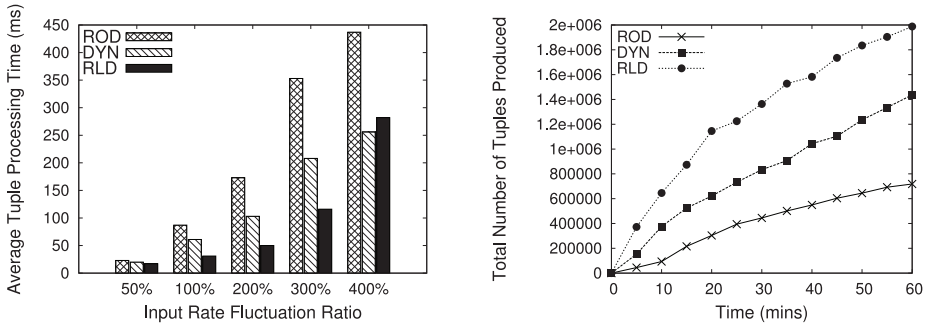
small. Compared to the *GreedyPhy* approach, *OptPrune* takes at most twice as much time to produce an optimal physical plan in worst case (shown in Figure 14(e) and (f)).

Figure 15 compares the physical plans produced by our *GreedyPhy* and *CorPhy* with the optimal solution produced by *OptPrune* for different queries (number of operators) for varying 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. As shown in Figures 15(c) and (f), none of the algorithms can completely cover the parameter space when the given resources (i.e., the number of machines) are not sufficient to cope with the full workload (i.e, the number of operators). The physical plan generated by *OptPrune* is always the optimal solution, yet the search costs are much cheaper than those of *ES*. Namely, the *OptPrune* and *ES* solutions and thus their quality are identical. Thus, *ES* is henceforth omitted (such as in Figure 15).

As for $rt_{GreedyPhy}$, the maximum rt is 0.94 and the minimum rt is 0.62 (the ratio varies with different queries). Clearly, *GreedyPhy* sacrifices the quality of the robust physical plan for a reduction in compilation overhead as indicated in Figure 14. Compared to *GreedyPhy*, *CorPhy* always achieves larger space coverage by analyzing the operator correlations among all given logical plans rather than considering one logical plan at a time. This results in 9% coverage gain on average compared to *GreedyPhy*'s solutions. Moreover, the *CorPhy* strategy returns the optimal robust physical plan approximately 60% of the time. Furthermore, the space coverage of *CorPhy* has an 11/9 OPT + 1 bound (where OPT is the optimal solution from *OptPrune*) from the First-Fit-Decreasing strategy for the deterministic bin packing problem [Chen et al. 2011].

7.5. 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 [Xing et al. 2006] (ROD) and dynamic load distribution [Xing et al. 2005] (DYN). The average tuple processing time essentially



(a) average tuple processing time (b) total number of tuples produced

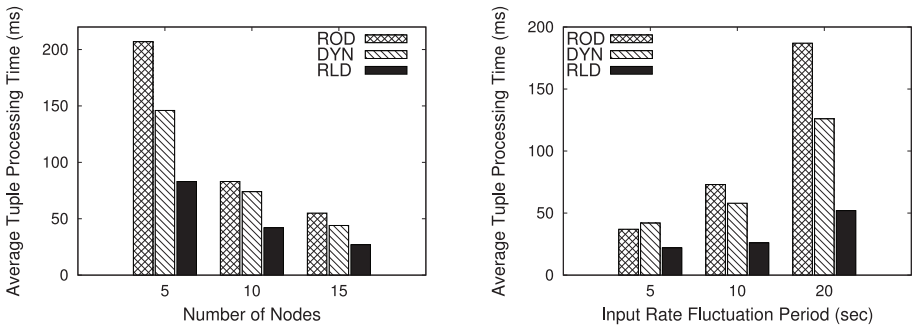
Fig. 16. RLD runtime performance (synthetic dataset).

corresponds to the end-to-end latency per tuple, which encompasses the impact that other factors have on the system's processing behavior. It shows how quickly our RLD system responds to fluctuating input streams. This is the main concern from the stream application users' point of view. Each query is run for 60 minutes five times using these different solutions with the initial setup as shown in Table IV. Using the synthetic datasets, we simulate dynamic changes as follows: the data generator starts with a chosen dataset and its initial input rate; over time, the input rates are scaled by a constant. This process is repeated continuously for infinite data streams.

In general, the results in Figure 16 show that RLD outperforms ROD and DYN in both metrics with the synthetic datasets. The primary reason is that neither ROD nor DYN guarantee any optimality of logical query plans. Rather, its load migration instead only focuses on changes of the operators' physical layout on computing nodes. Consequently, the query processing may still suffer from suboptimal plan execution (ordering) even after the load migration.

Total number of tuples produced. Figure 16(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 the three approaches to input stream fluctuations. We increase the input rates from 50% to 100% after the first 20 minutes and further increase the rates to 200% after running for 40 minutes. As indicated in Figure 16(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 suboptimal logical plan execution. Thus, our RLD approach performs better than DYN because RLD's robust logical plans can proactively smooth out the load changes.

Average tuple processing time. Figure 16(a) shows the average tuple processing time results over 60 minutes for the algorithms when the input rates vary from 50% to 400% of the initial rates as shown in Table IV. Specifically, the normal (100%) input rate serves as the single-point estimation for our RLD solution. The expected range of input rate fluctuations is set between 50% and 200% of the initial rate. The 300% and 400% of the initial rate correspond to scenarios when the input fluctuations exceed the expected range (i.e., they are not included in the parameter space). The results in Figure 16(a) over such wide range of fluctuations not only show that RLD is robust to the input rate variations in most cases, but also point out where it fails.



(a) vary the number of nodes (b) vary the input rate fluctuation period

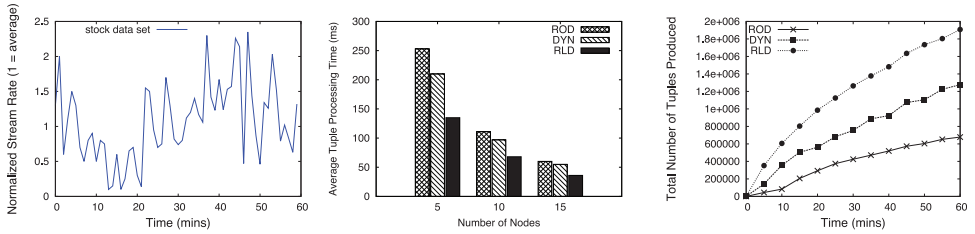
Fig. 17. RLD runtime performance (synthetic dataset).

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 scales well with the input rate. Our RLD approach performs 3 and 2 times better than ROD and DYN approaches respectively. This is likely due to ROD's performance suffering from executing suboptimal logical query plans once input data fluctuations arise. The 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 of DYN becomes 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 ratio exceeds the expected uncertainty level in the parameter space. Thus the load of the system cannot be well balanced since RLD only adopts 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 any solution that relies on one single physical allocation (as done by our RLD approach) to handle such fluctuations. Thus, this clearly denotes the scope of applicability for RLD, which not only requires fluctuations to be known a priori, but also to fit within some given range as expressed by the parameter space.

Our experiments further inspect whether the RLD approach is robust to two additional parameters, namely the number of nodes and the input stream fluctuation periods using the synthetic datasets. 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.

As indicated in Figure 17, both ROD and DYN do not exhibit the same robustness to these two parameters as RLD does. In particular, they do not perform well especially when the number of nodes is small (Figure 17(a)), or the load fluctuation period is long (Figure 17(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 three 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 proactively chooses between suitable logical plans



(a) input rate of stock dataset (b) robustness under changing resources (c) total number of tuples produced (10 nodes)

Fig. 18. RLD runtime performance (stock dataset).

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 somewhat increases while ROD and DYN suffer more for the long fluctuation periods. The reason is that ROD neither supports load migration nor has multiple logical plans corresponding to different load fluctuations. On the other hand, the DYN approach supports load migration, however, its performance is 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.

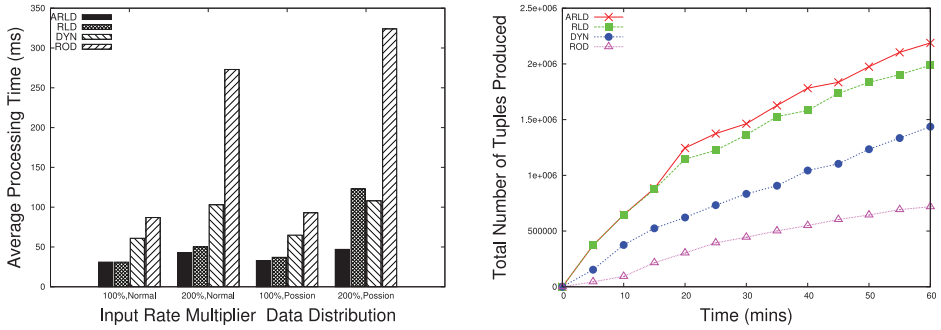
Performance with the real-life dataset. As depicted in Figure 18, our RLD approach also outperforms ROD and DYN with the real-life dataset (stock).

Figure 18(a) depicts the arrival rate of our stock dataset. In Figure 18(b), when the number of nodes is 5, our RLD approach gains 87% and 55% performance improvements over ROD and DYN, respectively. The performance difference gradually becomes smaller with an increase in the number of nodes. Again, when the number of nodes is small (5 nodes), the DYN approach migrates frequently. When the number of nodes increases (10 nodes) and thus the load per node decreases, the load migration becomes less necessary and thus infrequent. Eventually no migration is needed for the DYN approach in the case of 15 nodes. Thus, all three systems succeed to produce stable solutions. Our RLD approach still stands out because it proactively chooses between suitable logical plans at runtime.

We also evaluate the performance of our RLD solution on the real-life stock dataset with respect to the total number of tuples produced. The results shown in Figure 18(c) again confirm the robustness of our approach to stream fluctuations. In particular, our RLD approach produces tuples more smoothly compared to DYN, because our RLD requires no load migration when input streams fluctuate.

Handling space drift. Now we illustrate the effectiveness and efficiency of our newly proposed ARLD technique. In this experiment, we simulate the space drift by controlling not only the input rates but also the data distributions. 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 t_{rate} of the high-rate interval (i.e., the period of the input stream fluctuation) equals the duration of the low-rate interval. The data distribution change is simulated by switching between two synthetic datasets, following normal and Possion distributions, respectively. The duration of each data distribution t_{dst} equals $2t_{rate}$.

As indicated in Figure 19, our ARLD handles the space drifting better than the other approaches in both metrics. Specifically, ARLD performs well when the load fluctuation is high (Figure 19(a)). When the input rate is low and the dataset follows



(a) average tuple processing time (b) total number of tuples produced

Fig. 19. Handling space drift.

a uniform distribution, as expected, the performance difference among all four approaches is small. This is because when the single-point estimation is closed to the runtime statistics, all four systems succeed to produce stable solutions that require few or no operator migrations. However, when we change the dataset and increase the input rate, our ARLD significantly outperforms the other three because it is able to capture the space drift and produces a new RLD solution to adapt it at runtime. The new RLD solution provides better performance than any other solutions being used by the other approaches. Thus ARLD is capable of reducing the average processing time when the statistic significantly changes (i.e., space drift) by switching to the newly optimized solution.

As indicated in Figure 19(b), our ARLD quickly adjusts to the new input rates and the new data distribution. ARLD is quite effective at detecting and adapting to space drifts. It continues to produce results with the appropriate robust load distribution solution, either choosing the best logical plan or migrating to a new solution. On the contrary, ROD suffers from its static distribution limitation and barely produces any results when the input rate increases and the data distribution changes. While RLD still can keep up with the drifted space, it is incapable of reacting to this significantly different space. The logical plans being executed may no longer be robust with respect to the new space. Lastly, DYN is able to react to the space drift by migrating to a new physical layout. However, its performance is limited by its suboptimal logical plan execution.

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 reoptimization 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.

8. RELATED WORK

Robust query optimization [Babu et al. 2005; Markl et al. 2004] and parametric query optimization [D et al. 2008; Reddy and Haritsa 2005; Bizarro et al. 2009; Ioannidis et al. 1992; Graefe and Ward 1989] are closely related areas. Robust query optimization [Babu et al. 2005; Markl et al. 2004] 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 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 [Ioannidis et al. 1992; Graefe and Ward 1989] 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 [Reddy and Haritsa 2005; D et al. 2008] propose 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" to each other. PPQO [Bizarro et al. 2009] 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 preceding works [D et al. 2008; Bizarro et al. 2009]. 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 previous 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 probabilistic 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 [Zhu et al. 2004; Chandrasekaran et al. 2003; Abadi et al. 2003; Motwani et al. 2003] drew attention to distributed stream processing systems [Shah et al. 2003; Xing et al. 2005]. Flux [Shah et al. 2003] offers dynamic "intra-operator" load balancing by partitioning the input streams into substreams and determining the assignment of the substreams 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 [Xing et al. 2005] 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 precomputed (*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 [Kalyvianaki et al. 2011; Wolf et al. 2008]. SQPR [Kalyvianaki et al. 2011] 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 [Wolf et al. 2008] 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 proactively produce a robust load distribution solution *without* runtime operator migration. Unlike their effort, given known data fluctuations as input, we proactively 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 runtime), (b) query admission (evaluate expected resource needs of new queries and potentially restrict their execution), and (c) query migration (move operators across machines at runtime). None of these three topics is the focus of our work. Moreover, no 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 [Xing et al. 2006], which we compared against in our experimental study (Section 7.5), 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. First, 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. Second, 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 a 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. Third, 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 considers the uncertainty in both selectivities and input rates and their impact on query processing robustness. Consequently, our work constructs a more general model to achieve a larger scope of applicability of the produced robust load distribution solution.

9. 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 can 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 robust logical plan generator ERP then efficiently produces a robust logical solution that covers the parameter space. Taking the robust logical solution as input, our robust physical plan generator $OptPrune$ produces an optimal robust physical plan that supports the logical solution at runtime without operator relocation. Due to our effective bounding strategy, $OptPrune$ 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. We also show that the optimization costs of our robust load distribution solution are reasonable. Thus, this technique is well suited to modern DSPSs.

The work presented here focuses on highly robust load distribution without load migration. A possible direction for future work is to relax this constraint, and to move toward a hybrid solution in which load migration is considered when either stream data fluctuations go beyond the expected uncertainty level of data stream statistics or it is determined that no robust solution exists that can handle the full scope of fluctuations. This will radically change the optimization algorithms. We believe that, by supporting dynamic load migration, our current technique can be thought of as a way to minimize the frequency and cost of migration.

REFERENCES

Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The design of the borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'05)*.

Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, et al. 2003. Aurora: A new model and architecture for data stream management. *VLDB J.* 12, 2, 120–139.

David M. Allen. 1968. Handbook of methods of applied statistics. *Technometrics* 10, 4, 872–873.

Brian Babcock and Surajit Chaudhuri. 2005. Towards a robust query optimizer: A principled and practical approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. 119–130.

Shivnath Babu, Pedro Bizarro, and David Dewitt. 2005. Proactive re-optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. 107–118.

Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive ordering of pipelined stream filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. 407–418.

Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. 2004. Contract-based load management in federated distributed systems. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. 15–15.

Dimitris Bertsimas and Omid Nohadani. 2010. Robust optimization with simulated annealing. *J. Global Optim.* 48, 2, 323–334.

Pedro Bizarro, Nicolas Bruno, and David J. Dewitt. 2009. Progressive parametric query optimization. *IEEE Trans. Knowl. Data Engin.* 21, 4, 582–594.

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Helerstein, et al. 2003. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. 668–668.

Surajit Chaudhuri, Arnd Christian Konig, and Vivek Narasayya. 2004. SQLCM: A continuous monitoring framework for relational database engines. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*. 473–485.

Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. 2010. Variance aware optimization of parameterized queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 531–542.

Ming Chen, Hui Zhang, Ya-Yunn Su, Xiaorui Wang, Guofei Jiang, and Kenji Yoshihira. 2011. Effective vm sizing in virtualized data centers. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*. 594–601.

Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Cetintemel, et al. 2003. Scalable distributed stream processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'03)*. 257–268.

Thomas M. Cover and Joy A. Thomas. 1991. *Elements of Information Theory*. Wiley-Interscience, New York.

Harish D, Pooja N. Darera, and Jayant R. Haritsa. 2008. Identifying robust plans through plan diagram reduction. *Proc. VLDB Endow.* 1, 1, 1124–1140.

Ralf Diekman and Robert Preis. 1999. *Load Balancing Strategies for Distributed Memory Machines*. Civil-Comp Press, 124–157.

Goetz Graefe and Karen Ward. 1989. Dynamic query evaluation plans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'89).* 358–366.

Kien A. Hua, Yo Lung Lo, and Honesty C. Young. 1993. Considering data skew factor in multi-way join query optimization for parallel execution. *VLDB J.* 2, 3, 303–330.

Yannis E. Ioannidis and Stavros Christodoulakis. 1993. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. Database Syst.* 18, 4, 709–748.

Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1992. Parametric query optimization. *VLDB. J.* 6, 2, 132–151.

Navin Kabra and David J. DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98).* 106–117.

Evangelia Kalyvianaki, Wolfram Wiesemann, Quang Hieu Vu, Daniel Kuhn, and Peter Pietzuch. 2011. SQPR: Stream query planning with reuse. In *Proceedings of the International Conference on Data Engineering (ICDE'11).* 840–851.

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

Chuan Lei, Elke A. Rundensteiner, and Joshua D. Guttman. 2013. Robust distributed stream processing. In *Proceedings of the International Conference on Data Engineering (ICDE'13)*. 817–828.

Richard J. Lipton, Jeffrey F. Naughton, Donovan A. Schneider, and Sridhar Seshadri. 1993. Efficient sampling strategies for relational database operations. In *Proceedings of the Selected Papers of the 4th International Conference on Database Theory (ICDT'93).* Elsevier Science Publishers Ltd., 195–226.

Andrea Lodi, Silvano Martello, and Michele Monaci. 2002. Two-dimensional packing problems: A survey. *Euro. J. Oper. Res.* 141, 2, 241–252.

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, and Hamid Pirahesh. 2004. Robust query processing through progressive optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04).* 659–670.

Thomas M. Mitchell. 1997. *Machine Learning* 1st Ed. McGraw-Hill, New York.

Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, et al. 2003. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'03).* 245–256.

Rimma V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. 2009. Self-tuning query mesh for adaptive multi-route query processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT'09).* 803–814.

Athanasios Papoulis. 1984. *Probability, Random Variables, and Stochastic Processes*. McGraw Hill.

Liping Peng, Yanlei Diao, and Anna Liu. 2011. Optimizing probabilistic query processing on continuous uncertain data. *Proc. VLDB Endow.* 4, 11, 1169–1180.

Naveen Reddy and Jayant R. Haritsa. 2005. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05).* 1228–1239.

Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03).* 25–36.

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

Timothy M. Sutherland, Bin Liu, Mariana Jbantova, and Elke A. Rundensteiner. 2005. D-CAPE: Distributed and self-tuned continuous query processing. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05).* 217–218.

Feng Tian and David J. DeWitt. 2003. Tuple routing strategies for distributed eddies. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03),* vol. 29. 333–344.

TradingMarkets. 2013. http://www.tradingmarkets.com/.

Haixun Wang and Jian Pei. 2005. A random method for quantifying changing distributions in data streams. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'05).* 684–691.

Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. 2008. 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).* 306–325.

Ying Xing, Jeong-Hyon Hwang, Ugur Cetintemel, and Stan Zdonik. 2006. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06)*. 775–786.

Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. 2005. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. 791–802.

Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. 2004. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. 431–442.

Received April 2013; revised February 2014; accepted March 2014