Abstract

Cost-based query optimizer chooses the most efficient execution plan for a given query using a cost model. The latter relies on the accuracy of estimated statistics. These estimates often differ significantly from those encountered during query execution, leading to poor plan choices. In this document, we present a method to query processing that is fully aware of estimation inaccuracies. This method produces execution plans that are likely to perform reasonably well over different run-time conditions, so called robust plans. Robust plans are then augmented with extra-operators. These operators collect statistics at run-time and check the robustness of the current plan. If the robustness is violated, extra-operators are able to make decisions for plan modifications to correct the robustness violation without a need to recall the optimizer. We present the results of performance studies of our method, which indicate that it provides significant improvements in the robustness of query processing.

An Hybrid Method to Robust Query Processing With Respect to Estimation Errors

Chiraz Moumen Franck Morvan Abdelkader Hameurlain

January 20, 2017

0.1 Introduction

Cost-based query optimizers use cost models to determine the best among candidate execution plans for a given query. This plan has the lowest estimated cost [1]. Determining the best plan for a query requires accurate estimates of the cost model inputs (e.g., sizes of temporary relations). Unfortunately, these estimates are often significantly in error with respect to values encountered at run-time. Estimation errors can occur due to the use of outdated statistics, invalid assumptions (e.g., defaults values, attribute value independence), or because of the lack of sufficient information about the run-time conditions at compile-time [2]. Ioannidis et al. [3] showed that estimation errors propagate exponentially with the number of joins, leading to sub-optimal plan choices.

Several researches aimed to find solutions to the problem of plan sub-optimalities caused by estimation errors. These researches include techniques for better quality of the statistical meta-data [4–8]. Although these techniques improve the precision of estimates, obtaining accurate estimates remains a challenge since it requires detailed and a prior knowledge about data (e.g., attribute values distributions) and run-time characteristics (e.g., system load).

Motivated by the difficulty of providing accurate estimates, a part of the database research community proposed new approaches for query optimization that are able to detect and recover from plan sub-optimalities caused by estimation errors. We distinguished two main approaches [9]. One approach called *Single Point-based Optimization* [10–13] selects the best execution plan for a given query using single-point estimates of statistics needed for cost computations. Then, it tracks statistics during the execution. If an estimation error is detected, a re-optimization of the rest of the plan is triggered to correct the resulting sub-optimalities. The optimization and the execution stages of processing a query may be interleaved many times.

In this approach, the proposed methods do not incorporate issues affecting re-optimization. At compile-time, no information about the uncertainty in the used estimates is provided. Estimates are treated as they were completely precise. This makes methods very oppor-

tunistic to plan re-optimizations. Moreover, in a highly unpredictable environment, when re-optimization is triggered, the optimizer may use new inaccurate estimates, resulting in other plan re-optimizations. This may induce significant overhead and thus performance regression.

To avoid this problem, an alternative approach called *Multi point-based Optimization* [14–20] was introduced. Methods in this approach aim to make query optimization process aware of the possibility of errors in estimates. This is ensured by the use of probability distributions or intervals of estimates around error-prone parameters. The use of multiple points instead of single-point estimates of statistics quantifies the optimizer uncertainty regarding the accuracy of used estimates. Furthermore, considering possible run-time values of parameters allows the optimizer to identify plans that are expected to generate good performance in different run-time conditions. These plans are said robust.

The performance of methods using probability distributions (e.g., [14, 15]) may be limited. These methods assume that they have precise information about data distributions. However, this is rarely the case (e.g., dynamic environment, complex correlation). As for methods using intervals of estimates (e.g., [17, 19, 20]), they assume that it is usually feasible to find a plan, which is robust over a whole interval. This assumption is not often valid. High uncertainty regarding the accuracy of used estimates involves a large interval. Finding a plan that is robust over a large interval is not always feasible. Plans can only be robust over parts of that interval. Moreover, the majority of methods in this approach can be described as fully proactive. They anticipate the reaction to a potential errors in estimates and generate appropriate execution plans. These plans are kept unchanged until the termination of the executions. Indeed, these methods assume that the run-time values of parameters inevitably match the compile-time estimates of these parameters. This assumption is still unjustified and its use may result in poor performance. In this document, we focus on this issue.

We aim to extend this optimization approach, and particularly our work in [21], to make it able to respond to changes in the actual run-time conditions compared to run-time

conditions expected at compile-time. The method that we presented in [21] is part of the multipoint-based optimization approach. It divides -if required- an interval of estimates into sub-intervals and identifies plans, each of them is robust within a sub-interval. Then, it relies on a probabilistic reasoning to decide which plan to choose to start the execution. We suggested to calculate for each plan, its probability to avoid a robustness violation. The plan with the highest probability is selected.

Although the risk of robustness violation using this plan is low, a robustness violation may nevertheless occur and induce poor performance. As a solution, we propose in this document an hybrid method to query processing that is able to detect and correct a robustness violation during a query execution. A robust plan is firstly produced using intervals of estimates around uncertain parameters. At run-time, our extended method uses feedback from query execution to check the robustness of the current plan. A robustness violation can be detected and corrected through check-decide operators inserted into the plan at compile-time. These operators are inserted at key points. They are responsible for collecting statistics at run-time and deciding if a modification of the rest of the plan is necessary. They are able to react to a robustness violation autonomously, without a need to recall the optimizer. Decisions for plan modifications are prepared at compile-time and evaluated during the execution. To insert such operators, we rely on a user-defined parameter called risk threshold. A risk threshold indicates the maximum value of the risk of a robustness violation that the user is willing to accept. This threshold allows a trade-off between performance and uncertainty in used estimates.

The rest of the document is organized as follows: Section 2 provides the context of our work. Section 3 details our contribution. Section 4 presents the results of the performance evaluation of the proposed method. We discuss the most related work in Section 5. We conclude and present future work in Section 6.

0.2 Background and Motivating Example

In this section, we define first our concept of robust query execution plan. Then, we present an example that highlights our motivations.

0.2.1 Preliminaries

A solution to account for possible errors in estimates used by the optimizer, is to consider intervals around these estimates rather than specific values. The literature provides several techniques for computing such intervals (e.g., [10, 16, 22]). The computation of intervals of estimates is out of the scope of this document. To conduct our work, we relied on the method proposed in [22]. In [22], the authors presented a method for using upper and lower bound information to define intervals around uncertain parameters. Once computed, intervals of estimates are then used to identify execution plans that are expected to provide robust performance.

Basically, a plan is said robust if its performance does not regress significantly in the presence of estimation errors. To avoid confusion with other robustness concepts addressed in the prior literature, we propose the following definition: *let v_e be an estimate of an error-prone parameter value, let I be an interval of estimates around v_e exhibiting the uncertainty about this estimate. Let P_{best} be the best plan for a specific value $v_i \in I$. Finally, let λ be a user-defined cost-increase threshold (expressed in percentage). A plan P_{alt} is robust with respect to estimation errors if:*

$$\forall\ v_i \in I, \frac{cost(P_{alt})}{cost(P_{best})} \leq 1 + \frac{\lambda}{100} \tag{1}$$

For instance, if users tolerate a minor cost increase (λ) of 10%, the cost of P_{alt} is at most 1.1 times the cost of the best plan.

0.2.2 Motivating Example

The use of intervals fully quantifies the uncertainty regrading the accuracy of estimates. A large interval is a sign of high uncertainty, while a narrow interval is a sign of low

uncertainty. Generating a plan that is robust over a large interval is hard to achieve. The following example illustrates this issue.

Example 1. Consider the query Q:

Select ***

from *customer, order, item*

where *customer.ckey = order.okey* ***and***

customer.ckey = item.ikey ***and***

order.totalprice > [totalprice] ***and***

item.price < [price];

Q is a query joining the relations *customer*, *order* and *item*. There are a selection predicate on *order.totalprice*, denoted σ(order), of the form *order.totalprice > [totalprice]*, and a selection predicate on *item.price* of the form *item.price < [price]*, denoted σ(item). Suppose that |customer|=115MB, |order|=220MB, |item|=380MB, and that the database buffer size is 140MB. Throughout this document, the notation |R| refers to the size of a relation R.

Assume that an accurate estimate of |customer| is available from the database catalog, and that the cardinality of σ(order) is estimated accurately from an equi-height histogram on *order.totalprice* from the database catalog. Suppose that $|\sigma(\text{order})|$=120MB. As for $|\sigma(\text{item})|$, we assume that there is no histogram on *item.price* attribute. The cardinality of σ(item) is considered subject to an estimation error. To model this, an interval of estimates around $|\sigma(\text{item})|$, denoted I_i, is computed. Let I_i=[85, 170]MB. A query optimizer enumerates the plans P_{ic} and P_{co} (Cf. Figure 1), as two possible execution plans for Q. P_{ic} is the best plan for low values of $|\sigma(\text{item})|$. For higher values of $|\sigma(\text{item})|$, P_{co} becomes the best plan.

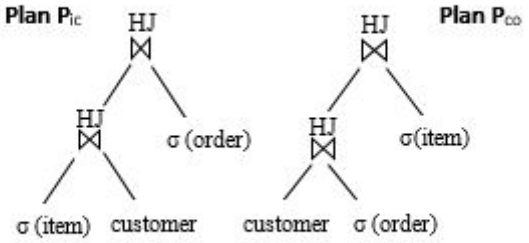


Figure 1: Execution plans for Q

Figure 2 below shows the variation of execution costs of these plans with respect to $|\sigma(\text{item})|$. This figure highlights the difficulty of producing a single robust plan over a large interval of estimates. Considering only the range [85, 145]MB, P_{ic} can be chosen as a single robust plan since its cost remains close to the best cost within this range. However, considering the interval [85, 170]MB, we notice that finding a single robust plan is not feasible since neither P_{ic} nor P_{co} is robust within this interval.

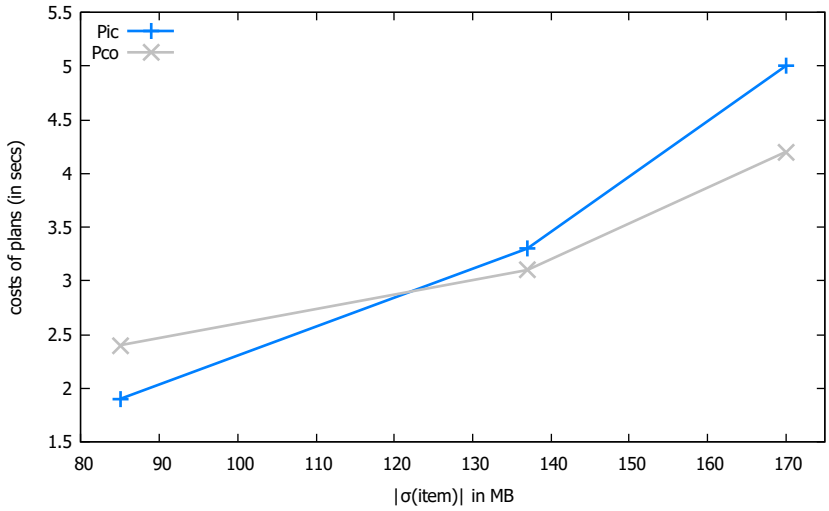


Figure 2: Costs of plans with respect to $|\sigma(\text{item})|$

We addressed this problem in details in [21]. We proposed an optimization method that divides an interval of estimates into sub-intervals and generates a set of plans, each

of them is robust within a sub-interval. Only one robust plan among those found must be then chosen to begin the execution. For that purpose, we relied on a probabilistic reasoning. We calculate for each plan its probability to avoid a robustness violation and choose the plan with the highest probability. A robustness violation occurs when the value observed at run-time does not lie within the robustness range of the chosen plan. The performance studies of this method showed that it improves the optimizer ability to generate robust plans, especially when the uncertainty is high [21].

The research work in [21] focuses exclusively on the compile-time choices of robust execution plans. Although a chosen plan has the highest probability to avoid a robustness violation, the risk that values observed at run-time are not belong to the robustness range of that plan well exists.

Going back to the example of the query Q in the subsection 2.2. As depicted in Figure 2, the robustness range of P_{ic} is [85, 145]MB, and that of P_{co} is [105, 170]MB. P_{co} has a higher probability to avoid a robustness violation. Its robustness range covers more values that are likely to be observed at run-time. P_{co} is chosen to begin the execution. If at run-time, $|\sigma(\text{item})|$ turns out to be 95MB, continuing to execute P_{co} will result in a robustness violation and thus performance regression. One solution would be to interact with the execution environment. Markl et al. [20] demonstrated that using feedback from query execution to verify the adequacy of the running plan can improve performance by orders of magnitude. Methods relying on this principle (e.g., [10, 20]) propose to use first a traditional optimizer to generate the best plan for a query. This plan is then augmented with extra-operators that are inserted at specific points in the plan. These operators collect updated statistics at run-time. These statistics are used to determine whether a re-optimization of the rest of the plan is necessary. These methods use heuristics to insert extra-operators. In addition, plan re-optimizations are supervised by the optimizer, which can induce a significant additional cost due to multiple re-invocations of the optimizer.

Unlike existing methods, we propose to insert what we call check-decide operators, at the points in the plan where there is a high level of uncertainty. This level is specified

based on a user-defined parameter. At run-time, statistics are collected at these points. Check-decide operators use information gathered at compile-time to react autonomously and correct a robustness violation without a need to re-invoke the optimizer. Details of this work are described in the next section.

0.3 Hybrid Method to Robust Query Processing

In this section, we detail our method called **Hybrid Method to Robust Query Processing (HRQP)**. This method addresses two problems: (1) the production of robust execution plans that handle the uncertainty in estimates used at compile-time, and (2) the detection and the correction of a robustness violation at run-time. The method includes three main modules:

- **Identification of Robust Plans:** the risk of errors in estimates used at compile-time is modelled by means of intervals computed around these estimates. The cost of possible execution plans are compared over these intervals and plans providing robust performance are identified. The less likely plan to result in a robustness violation during the execution is selected to start the execution.

- **Insertion of Check-Decide Operators:** these operators are inserted at specific points in the chosen plan. At run-time, up-to-date statistics are collected by these operators. Collecting statistics can cause a significant overhead if it is done at many points in a plan. To avoid that happening, at compile-time, the most effective points for collecting statistics are determined, and these operators are inserted into the plan at these points.

- **Query Plan Modification:** feedback from query execution is used to determine whether the rest of the execution plan requires modifications. Decisions for plan modifications are managed by the check-decide operators inserted into the plan at compile-time. These operators are able to made decisions autonomously, based on information gathered at compile-time and those collected at run-time.

In the remainder of this subsection, we detail each of the above modules. We continue to rely on the scenario of the example in the subsection 2.2 to describe how these modules interact with each other.

0.3.1 Identification of Robust Plans

This module consists in a compile-time strategy to identify robust execution plans for given queries. This strategy extends our earlier work in [21], which does not consider join ordering with respect to estimation errors.

Our extended strategy proceeds in two steps to construct a robust execution plan for a query: (1) the first step consists in specifying a robust execution order of query operators and generates what we call logical operator plan, (2) the second step consists in the selection of algorithms to implement each of the operators of the logical plan and produces what we call a physical operator plan.

The objective of the first step- which is the distinguished feature of our extended stategy for identifying robust plans- is to specify a logical operator order taking into account the uncertainty in the estimated sizes of operand relations. Recall that the cost of a binary operator (i.e., join) is a function of the sizes of the two operand relations. The distinction between these relations is crucial because some join cost formulas (e.g., nested loop join) are not symmetric with respect to the inner and outer relations [1]. The outer relation corresponds to the operand relation from which tuples are firstly retrieved by the join algorithm. The inner relation corresponds to the operand relation from which tuples are then retrieved using join values of the outer relation [1]. By convention, we consider that the outer relation is the left join-input and that the inner relation is the right join-input in the plan tree. We put the relation with the smallest size at the left. This process is repeated for each binary operator in the logical plan.

Finding the relation with the smallest estimated size is simple when single-points estimates are used. However, when sizes are modelled by intervals of estimates, we calculate for each relation its probability to have the smallest size as follows: *Let two relations T and*

S and two intervals I_T=[a, c] and I_S=[b, d] including possible run-time values of $|T|$ and $|S|$. The probability that $|T|$ is lower than $|S|$, denoted prob($|T| < |S|$), is calculated as follows:

$$prob(|T| \leq |S|) = \begin{cases} \frac{b-a}{c-a} + \frac{d-b}{2(c-a)} & \text{if } [b,d] \subset [a,c] \\ \frac{d-c}{d-b} + \frac{1}{2} & \text{if } [a,c] \subset [b,d] \\ \frac{1}{2} & \text{if } [a,c] = [b,d] \\ \frac{d-a}{2(c-a)} & \text{if } a \in [b,d] \text{ and } d \leq c \\ \frac{b-a}{c-a} + \frac{d-c}{d-b} + \frac{c-b}{2(c-a)} & \text{if } c \in [b,d] \text{ and } a \leq b \end{cases}$$

When only |T| is modelled by an interval I_T and that a single-point estimate of |S| is used, the probability that |T| is lower than |S|, is calculated as follows:

$$prob(|T| < |S|) = \begin{cases} 0 & \text{if } |S| < a \\ 1 & \text{if } |S| > c \\ \frac{|S|-a}{c-a} & \text{if } a \leq |S| \leq c \end{cases}$$

Explanation of the formula for computing these probabilities is available in the APPENDIX.

To describe this module, we consider the query Q in the subsection 2.2. We removed the equi-height histogram on attribute *order.totalprice* from the catalog. An interval of estimates around $|\sigma(\text{order})|$, denoted I_o, is computed. Let I_o=[75, 160]MB. Applying the rules above to our example with as inputs the single-point estimate of |customer| and the intervals I_i and I_o, the plan P_{log} (Cf. Figure 3) in the form of a logical-operators tree, is selected to execute Q. This plan is used as input to the second step.

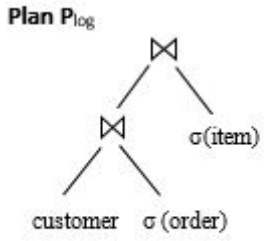


Figure 3: logical plan for Q with respect to I_i and I_o

The second step consists in applying to each operator in the logical plan the following process: physical algorithms are enumerated and their costs are compared over the estimated sizes of the operand relations. When a relation size is modelled by an interval of estimates, we calculate the robustness range of each candidate algorithm. For example, let O be the first logical operator in P_{log}, φ is a set of candidate physical algorithms for O. Suppose that an accurate estimate of |customer| is available from the catalog. However, $|\sigma(\text{order})|$ is considered subject to an estimation error. The interval I_o around $|\sigma(\text{order})|$ is used.

Calculating the robustness range I_{sub} for a physical algorithm O_{alt} in φ can be converted into a numerical solving problem. We compute the lower bound of I_{sub} by solving the inequality:

$$Cost(O_{alt}, |customer|, |\sigma(order)|) \leq (1 + \frac{\lambda}{100}) \times CostBestPlan(\varphi, |customer|, |\sigma(order)|) \quad (2)$$

$CostBestPlan(\varphi, |customer|, |\sigma(order)|)$ returns the cost of the best plan in φ for |customer| and $|\sigma(\text{order})|$. $Cost(O_{alt}, |customer|, |\sigma(order)|)$ returns the execution cost of O_{alt}.

Similarly, the upper bound of I_{sub} is determined by solving the inequality:

$$(1 + \frac{\lambda}{100}) \times CostBestPlan(\varphi, |customer|, |\sigma(order)|) \leq Cost(O_{alt}, |customer|, |\sigma(order)|) \quad (3)$$

Solving an inequality with $|\sigma(\text{order})|$ as the variable of the inequality consists in varing $|\sigma(\text{order})|$ in I_o, comparing the cost of O_{alt} with the best cost and finding the minimum root for which the inequality is true. To avoid a large computational complexity, the comparison of costs is done only at a few points in the intervals. These points are determined using our modified secant method [21]. We relied on the secant method because it provides precise results with reduced complexity. Moreover, it can be extended to n-parameters space [23]. More details on calculating robustness ranges are provided in [21]. After robust algorithms for each logical operator are identified, we rely on a probabilistic approach to choose for each operator the algorithm with which to start the execution. We compute for each algorithm its probability to avoid a robustness violation at run-time. Take as an example the physical algorithm O_{alt}, with a validity range I_{sub}. The probability to avoid a robustness violation choosing O_{alt}, denoted prob(O_{alt}) is calculated as follows:

$$prob(O_{alt}) = \frac{width\ of\ I_{sub}}{width\ of\ I_o} \tag{4}$$

This rule applies when only one of the operand-relations is modelled by an interval of estimates. Details about the generalized case can be found in [21].

0.3.2 Insertion of Check-Decide Operators

In the previous module, a logical plan for a given query is produced and robust algorithms are selected to implement the logical operators in that plan. The robustness of an algorithm is evaluated over intervals of estimates around operand-relations sizes. An algorithm can be robust over a whole interval of estimates or over only a sub-interval of that interval. In the latter case, a robustness violation may occur if values observed at run-time do not match those for which the chosen algorithm is presumed robust.
As a solution to this problem, we propose in this document to augment the robust plan initially produced with extras-operators called *check-decide operators*. These operators have the dual role of monitoring a plan execution and, if necessary, deciding to modify

the rest of the plan to correct a robustness violation. Inserting these operators at many points in a plan and checking the robustness of this plan at each point may induce a significant overhead. To avoid this, we propose to insert these operators at the points in the plan where the risk of a robustness violation is high. An uncertainty level, which may be either *low* or *high*, is assigned to the algorithms chosen to execute operators of a plan. A high uncertainty level indicates that there is a significant risk that a chosen algorithm results in a robustness violation.

An uncertainty level is assigned relying on a user-defined parameter, called *risk threshold*. A risk threshold defines the maximum value of the risk of a robustness violation that the user is willing to accept. Take as example an algorithm O_{alt} chosen to execute a logical operator O as it presents the highest probability to avoid a robustness violation. Consider that this probability is equal to 80%. Thus, the risk of a robustness violation during the execution of that algorithm is 20%. If the risk threshold is 30%, the choice of O_{alt} is considered sufficiently reliable, the uncertainty level attributed to this algorithm is low. On the other hand, the use of a risk threshold of 10% would result in a high uncertainty level for O_{alt}.

A check-decide operator is inserted into an execution plan before each physical operator whose associated risk is considered high (Cf. Algorithm 1).

```
1 BEGIN;
2 for each operator in the execution plan ;
3       attribute uncertainty level ;
4       if (level='high') then ;
5             insert check-decide operator in the plan edge where the uncertainty
    should be resolved ;
6 END
```

Algorithm 1: Insertion of check-decide operators

At run-time, up-to-date statistics are collected at these points and a modification of the rest of the plan is triggered if required. We detail this in the subsection below.

0.3.3 Query Plan Modification

In this subsection, we describe how statistics can be collected at specific points during the execution of a query plan. We describe the types of statistics we can collect, and how this can be done while avoiding significant overhead. These statistics are then used to obtain improved estimates for intermediate result sizes and operator execution costs.

As a first step, we describe the method of collecting the statistics. Then we discuss the question of determining which statistics to collect and to which points of the execution plan of the query.

Consider the figure 4. There is a selection operator that applies to the *item* relation. Immediately after the selection operator, a check-decide operator can be inserted into the execution plan of the query to collect statistics. When the tuples are produced by the selection operator, they can be examined by the control operator and the required statistics can be collected without interrupting the execution of the plan; The cardinality of the selection operator can be calculated by maintaining a current count of the number of tuples that follow in the control operator.

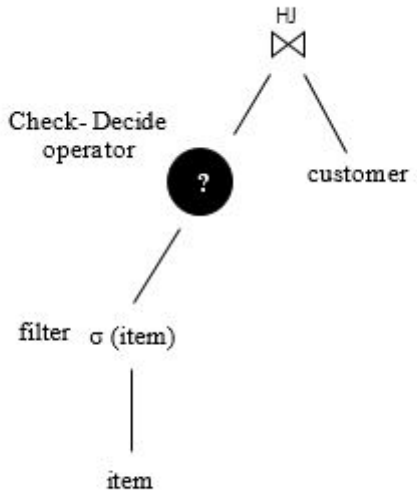


Figure 4: Statistics Collection during the execution

This approach to statistical gathering makes it possible to obtain updated statistics on cardinalities with reasonable accuracy. However, it has a major limitation, which concerns the pipeline of operators when executing a query. If statistics are gathered in the midst of pipeline execution of operators, none of the operators of the pipeline can benefit from these statistics. Indeed, all the operators of the pipeline run at the same time as the collection of statistics. Consequently, the statistics will only be ready when all the operators of the pipeline have completed a significant part of their execution.

An alternative to this would be to use the approach to materialisation of intermediate results. It should be noted that in the case of a pipeline, the pipeline will be interrupted at the points where the statistics are to be collected. This can slow down the execution of a query.

In our method, we do not address this problem. It is an interesting area that would be the subject of our future work.

In this document, we determine the points in an execution plan where a collection of statistics is likely to be beneficial for a robust execution of the query. As described in the previous section, this responsibility is delegated to the user, who specifies using the "risk threshold" parameter the desired trade-off between the correction of the plan during its execution and the final time of the query processing.

Once the statistics are collected in this way during query execution, they can be exploited to obtain new estimates for intermediate result sizes and operator execution costs for the rest of the query.

Check-decide operators are inserted into a query plan to ensure the robustness of this plan during the execution. When inserting these operators, the information concerning candidate physical algorithms for each operator along with their robustness ranges are incorporated into the corresponding operator.

At run-time, updated statistics are collected by the check-decide operators. If these statistics rely within the robustness range of the running operator, the execution continue without any modification. Otherwise, the current execution is interrupted and a modifi-

cation of the the plan is triggered.

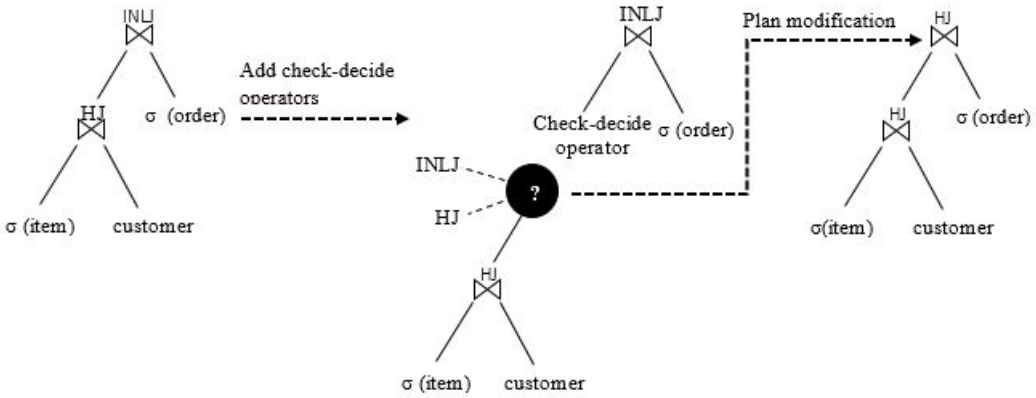


Figure 5: Modification of a query execution plan

Decisions for plan modifications are made by the check-decide operators (Cf. Figure modif). The latter can react in an autonomous manner on the basis of information gathered at compile-time. This avoids re-invoking the optimizer to re-optimize the rest of the query plan during the execution, and thus dodges the overhead resulting from re-calling the optimizer multiple times.

0.4 Experiments

In this section, we present our experimental framework and the results of the performance evaluation of our method. We compare the performance obtained by our method, denoted HRQP, with those obtained by the methods in [10] and [16]. The method proposed in [10], denoted DRO, relies on the single-point based optimization approach. The method in [16], denoted RIO, uses the multipoint based optimization approach. We study the behaviour of each of these methods by varying the error in the estimates of the intermediate relation

sizes as well as the uncertainty level. This latter may be high, medium, or low.
We conducted our experiments using a simulation model that we describe below.

0.4.1 Simulation Model

To carry out our experiments, we used a query builder and a simulator. The query builder generates a series of 100 queries and selects relations whose sizes are considered subject to estimation errors.
The simulator includes the proposed method and a database catalog: the simulated method consists mainly of two modules: 1) a query optimizer, and 2) a query executor.
The database catalog includes information about the query processing environment (e.g., CPU performance, available memory amount) as well as information describing data (e.g., number of tuples in relations, cardinality of attributes). The main simulation parameters are summarized in Table 1.

Table 1. Summary of dataset used in the experiments

Parameter	**Value**
Buffer size	400 MB
CPU performance	100 000 MIPS
Disk bandwidth	100 MB/s
Average disk latency	20 ms
Size of a page on disk	4 KB
Size of a record	1 - 3 KB
Size of a relation	100 - 1000 MB
Size of an attribute	10 - 500 Bytes

For all experiments, we considered that the cost-increase threshold for the robustness condition is 20%. This choice is motivated by the works of [16] and [17]. Recall that we used the method proposed in [22] to calculate intervals of estimates. Note that the higher is the uncertainty, the larger are the intervals. Intervals of estimates are used by HRQP

and RIO to generate robust execution plans for input queries. As for DRO, it makes use of specific estimates to choose the presumed least costly among candidate execution plans. In the next subsections, we present the results of our experiments.

0.4.2 Experimental Results

We seek through our experiments to visualize the impact of the risk threshold, the errors in the estimated sizes of operand-relations, and the uncertainty level on the quality of query processing.

0.4.2.1 Impact of risk threshold on execution times of our method

The first experiment consists in studying the impact of the user-defined risk threshold, denoted RT, on the performance of our method. Figure 4 shows the median value of the execution times of queries for different settings of the risk-threshold. Each of the curves in Figure 4 plots the variation of execution times with respect to errors on the operand-relations sizes, for a fixed value of the risk-threshold.

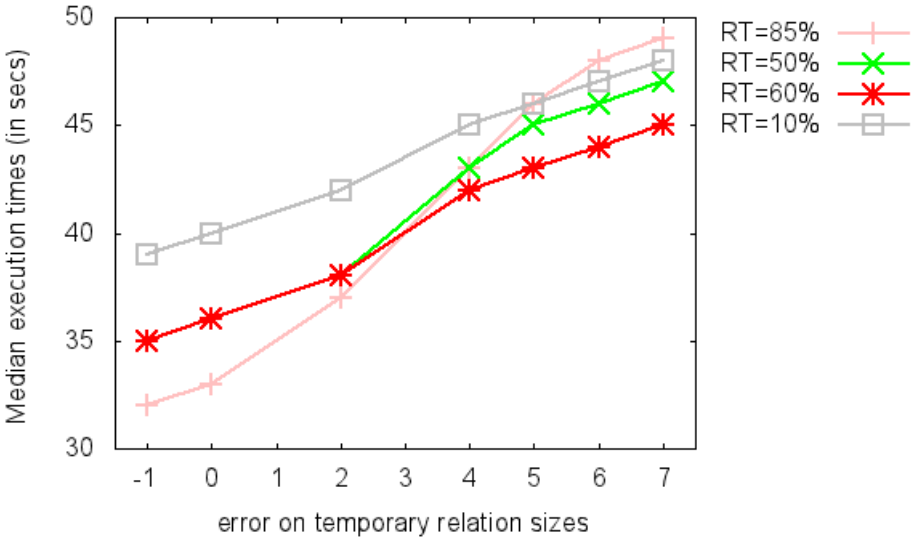


Figure 6: Impact of risk threshold on execution times

Given the result shown in Figure 4, we observe that extreme values of the risk threshold

can induce poor performance. This is because low values of the risk threshold (e.g., RT=10%) leads to the insertion of many check-decide operators. This may be beneficial when the error is important. However, checking the robustness and comparing statistics observed with those expected at many points in an execution plan can cause additional overhead if the error is low. Conversely, using higher values of the risk threshold (e.g., RT=85%) leads to the insertion of only few check-decide operators. Consequently, some robustness violations can not be detected and corrected. This induces poor performance when the error is important.

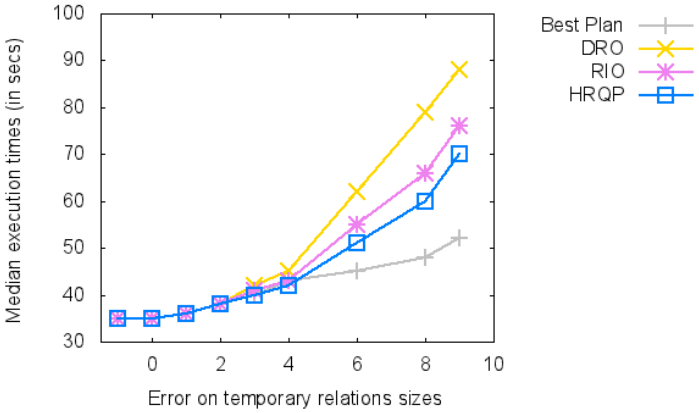
Figure 4 gives us a way to visualize the various performance that can be achieved by picking a particular risk threshold. A second observation suggested by Figure 4 is that moderate settings of the risk threshold are better than extremely high or low settings at producing acceptable execution times. Our experiments show that the best median execution time occurs when the confidence threshold is 60%. For this reason, the risk threshold is held constant at 60% in the remainder of our experiments.

0.4.2.2 Impact of estimation errors on execution times of methods

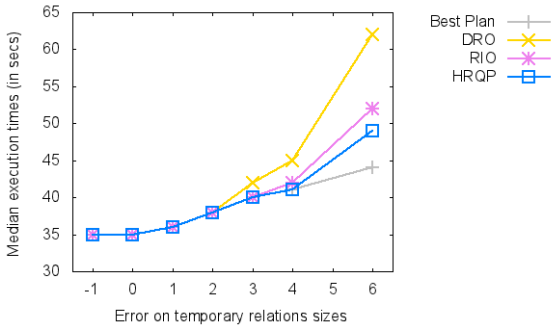
This experiment consists in evaluating the impact of estimation errors on the execution times of methods. Figure 5 illustrates the variation of execution times of plans produced by DRO, RIO and HRQP with respect to error on temporary relation sizes. Figure 5 also shows the execution times of the best plans.

The execution time by a method is calculated as the median of the execution times of all queries in the series by this method. We decided to use the median rather than the average value because extreme execution times can sharply draw up/down the average value and thus give a vision that is not representative of the majority of values. The error represented on the x-axis is calculated as the quotient resulting from the division of the actual size of a temporary relation by the size estimated at compile-time [16]. We apply the same error to each temporary relation in the plans of experimental queries. A positive error indicates an underestimation while a negative error indicates an overestimation of

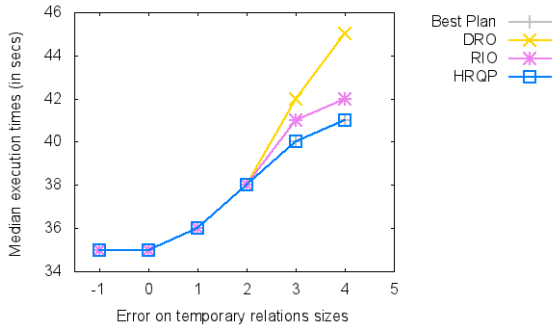
the value observed at run-time compared to that estimated at compile-time.



(a) High uncertainty



(b) Medium uncertainty



(c) Low uncertainty

Fig. 5. Variation of execution times with respect to errors on temporary relation sizes

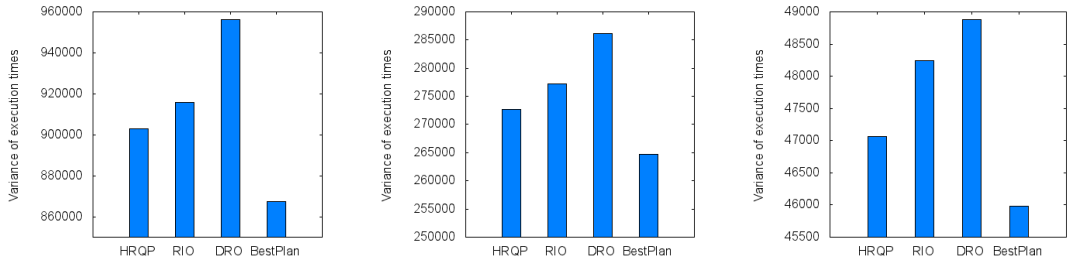
In Figure 5, we observe that DRO provides performance that are close to the best performance when the error is very low. However, when the error becomes significant (i.e., greater than 2), there is a significant increase in the execution costs of plans produces by DRO. Indeed, DRO generates execution plans whose costs are optimal for specific values of the temporary relations sizes. At run-time, if the observed values do not match those expected, the query optimizer is re-invoked to correct the resulting sub-optimalities. During re-optimization, the optimizer uses new specific values and produces the best plan for the rest of the given query. These values may be further erroneous leading to new plan re-optimizations. This can happen several times during processing a query. Consequently, it induces an additional cost.

As re-optimization is a costly operation, RIO seeks to reduce the need to re-optimize. It produces execution plans that are able to generate robust performance for different values of temporary relations sizes. These plans are identified over intervals of estimates calculated around the single-estimated sizes of these relations. Figure 5 shows that RIO has lower execution times than DRO. We also notice in Figure 5 that the lower is the uncertainty, the more are close the execution times by RIO and HRQP. Indeed, when the used intervals are narrow, it's feasible for RIO and HRQP to find a single robust plan over this interval. RIO and HRQP perform similarly and produce close performance. However, execution times by RIO are greater than those by HRQP when the uncertainty becomes important. Indeed, it becomes difficult to find a single robust plan over intervals. In this case, RIO seeks to identify a set of plans, said *switchable plans* [16]. The suitable plan is then chosen at run-time. The difference between execution times by RIO compared to HRQP is due to the fact that when RIO fails to generate a set of switchable plans, it behaves like the method in [20]. It considers specific estimates of statistics and chooses an optimal plan. During the execution, it tracks actual statistics by means of *check operators* inserted into the plan at compile-time. A re-optimization of the rest of the plan is triggered when the plan is no longer optimal. As with DRO, the method in [20] can result in suboptimal performance due to multiple plan re-optimizations.

Unlike RIO, HRQP always relies on multipoint estimates of statistics. When there is not a single robust plan over an interval, HRQP divides this interval into sub-intervals and generates a set of plans, each of them is robust within a sub-interval. Then, HRQP selects the plan with the highest probability to avoid a robustness violation to start the execution. At run-time, HRQP monitors the execution by means of check-decide operators inserted into this plan at compile-time. If a robustness violation is detected, these operators are able to react based on the information gathered at compile-time. A new robust plan is chosen for the rest of the plan without a need to recall the optimizer. This concept along with robustness sub-interval substantially reduce the run-time of queries that were optimized using single erroneous estimates.

0.4.2.3 Impact of estimation errors on the consistency of methods

The consistency of a method denotes its ability to cope with changes in the run-time conditions compared with those expected at compile-time. A method has a high consistency if its performance does not deteriorate significantly in the presence of estimation errors. Note that the consistency is inversely proportional to the variance [24]. In order to measure the variance of DRO, RIO and HRQP, we compute the variance of the performances of each method and compare it with the variance of optimal performances. A high variance indicates a significant dispersion of execution times. For this experiment, we use the same queries as before. The experimental results are illustrated in Figure 6.



(a) High uncertainty (b) Medium uncertainty (c) Low uncertainty

Fig. 6. Variance of execution times with respect to errors on temporary relation sizes

Figure 6 confirms that HRQP provides more stable performance compared with DRO and RIO. This figure shows that the variances of RIO and HRQP are close when the uncertainty is low. However, the greater is the uncertainty, the greater is the gap between their variances.

0.5 Related Work

The problem of performance penalty in query processing caused by estimation errors has been the subject of several researches. In this section, we compare our work with existing methods, the most related being [10, 16, 17, 19, 20, 25, 26]. Like in this document, these methods use intervals of estimates to generate appropriate execution plans. Nevertheless, our work differs from these methods on several aspects.

In the initial optimization phase in Rio [16], the uncertainty of a parameter is modelled by an interval of estimates around a single-point estimate of this parameter. If the plan chosen by the optimizer at the lower and the upper bounds of the interval is the same as that chosen at the single-point estimate, then this plan is resumed robust over the whole interval. This definition of robustness is still unjustified. In our method, the robustness of a plan over an interval of estimates is checked at multiple points in this interval. These points are determined using the principle of the secant method, which provides precise results with reduced complexity. Moreover, Rio [16] assumes that at least one of the input-join relations is a base relation. Contrary to this, our method can be applied to the general case where the sizes of both join-input relations are modelled by intervals of estimates.

The use of intervals may appear similar to the principle of parametric optimization [17, 19, 25, 26]. In this optimization approach, execution plans for a query are enumerated at compile-time such that each plan is optimal for a range of possible parameter values. The complexity of these methods lies in the large number of plans to be generated, stored, loaded, and compared at run-time. We reduce this complexity by seeking for robustness rather than conditional optimality. Accepting a minor increase in plan costs compared to the optimal cost reduces the number of plan to be generated and compared without degrading performance. In addition, a parametric optimization is particularly interesting in the case of queries that are compiled once and executed many times with minor modifications of the parameters. Moreover, existing methods do not consider the collection of statistics during execution, the chosen plan is used to execute the query until termination. In our work, a plan initially chosen to execute a query can be modified if a robustness violation is detected during execution. This is carried out by means of extras-operators, inserted into a query plan at specific points. The use of such operators can remind the work of [10] and [20].

In [10] and [20], the authors introduced algorithms to detect plan suboptimalities during executions through check operators inserted into the plans at compile-time. As in our

work, these operators collect statistics at run-time to check whether a correction of the rest of the plan is necessary. Unlike these methods, which use heuristics to determine the points where to collect statistics, we propose to collect statistics only at the points in the plan where the uncertainty level is considered high. This level is determined based on a user-defined parameter. This parameter allows a trade-off between performance and uncertainty in used estimates. At run-time, if a suboptimality is observed, [10] and [20] suggested to recall the optimizer to re-optimize the rest of the plan. This operation can be very costly if performed several times during an execution. In our method, it is no longer up to the optimizer to make plan changes. These are the check-decide operators who decide on the modifications of the rest of the plan. Decisions are prepared at compile-time and evaluated at run-time.

In [10], the uncertainty in estimates is ignored at compile-time. The optimizer uses specific values of the cost model parameters to choose a query execution plan. In [20], the authors extended the method in [10]. They introduced the use of what they called *validity ranges*. The use of validity ranges may seem similar to our work. There is, however, one major difference. In [20], an optimal plan is initially produced using traditional optimizer. Then, one or more extras-operators are added to this plan. Each operator has a condition that indicates the cardinality bounds, called *Validity ranges*, within which the plan is valid. The objective in [20] is to reduce the need of re-optimizations, but not to ensure robustness. Our objective is to produce execution plans that are robust for different execution conditions rather than plans providing optimal performance in particular execution conditions.

0.6 Conclusion and Future Work

This document presented a query processing method that is able to produce robust execution plans, detect and correct a robustness violation during the execution. In our work, a plan is said robust if it generates acceptable and stable performances in the presence of

estimation errors. Our method uses what we called check-decide operators to detect and correct a robustness violation at run-time. These operators are inserted into a plan at points where there is a high uncertainty level. If necessary, these operators make decisions for plan modifications without a need to recall the optimizer. The performance evaluation shows that our method provides more stable performance, especially in the presence of large estimation errors and thus improves the robustness of query processing, compared to the main existing methods.

The proposed method focuses on the errors in estimated sizes of temporary relations produced at intermediates stages in a plan. As future work, it would be interesting to extend our method to deal with more error-prone parameters. Another area for future work would be to extend our method, which actually performs in a uniprocessor environment, into a parallel environment and to study the impact of the parallelism degree choices over the robustness of query processing.

0.7 APPENDIX

0.7.1 A

Let R and S be two random variables following an uniform probability distribution over the respective intervals I1 = [a, c] and I2 = [b, d]. We try to calculate the probability that R is less than S, denoted P ($R \leq S$). We calculate this probability in the different possible cases:

1. **Case 1: b $\in$ [a, c] et d>c :**

   $P(R \leq S) = P[(a \leq R \leq b \cup c \leq S \leq d) \cup (b \leq R \leq c \cap R \leq S \leq c)]$

   To carry out this calculation, we use the following probability distribution: *let A and B be two events, P(A $\cup$ B) = P(A) + P(B) - P(A $\cap$ B)*

   if A and B are independent from each other, then P(A $\cap$ B) = P(A) $\times$ P(B)

if A and B are two incompatible events, then P(A ∩ B) = 0

We apply this rule to our case, that is A : (a ≤ R ≤ b ∪ c ≤ S ≤ d) and B : (b ≤ R ≤ c ∩ R ≤ S ≤ c). A and B are incompatible, we obtain:
P(R ≤ S) = P(a ≤ R ≤ b ∪ c ≤ S ≤ d) + P(b ≤ R ≤ c ∩ R ≤ S ≤ c)

Suppose now that A : (a ≤ R ≤ b), and B : (c ≤ S ≤ d).
A and B are independent, we obtain:
P(R ≤ S) = P(a ≤ R ≤ b) + P(c ≤ S ≤ d) - P(a ≤ R ≤ b) × P(c ≤ S ≤ d) + P(b ≤ R ≤ c ∩ R ≤ S ≤ c)

In probability theory, the probability that a random variable X is in an interval [a, b] can be calculated using the probability density function associated with this variable. A probability density, denoted *f*, is a function which allows to represent a probability law in the form of integrals. We say that a function f is a probability density of a random variable X if:

$$\forall\ x,\ P(X \leq x) = \int_{-\infty}^{x} f(t)\mathrm{d}t$$

The probability P(a ≤ X ≤ b) is calculated as following:

$$P(a \leq X \leq b) = \int_{a}^{b} f(t)\mathrm{d}t \quad \forall\ a < b$$

The density of probability f is defined as:

$$f(t) = \begin{cases} \dfrac{1}{b-a} & \text{if } a \leq t \leq b \\ 0 & \text{otherwise} \end{cases}$$

Let us return to oue example. Suppose that there is a density function on the variable R within I_1, denoted f_R and a density function over the variable S within I_2, denoted f_S:

- $P(a \leq R \leq b) = \int_a^b f_R(t)dt = \int_a^b \frac{1}{c-a}dt = \frac{1}{c-a} \times [t]_a^b = \frac{b-a}{c-a}$
- $P(c \leq S \leq d) = \int_c^d f_S(t)dt = \int_c^d \frac{1}{d-b}dt = \frac{1}{d-b} \times [t]_c^d = \frac{d-c}{d-b}$

We now compute $P(b \leq R \leq c \cap R \leq S \leq c)$:

- $P(b \leq R \leq c \cap R \leq S \leq c) = P(R \leq S,\ b \leq R \leq c,\ b \leq S \leq c)$
$= \int_b^c P(R{=}t,\ S{¿}t)\ dt = \int_b^c \frac{1}{c-a} \times \frac{c-t}{c-b}dt$
$= \frac{1}{(c-a)(c-b)} \times \left[\frac{-(c-t)^2}{2}\right]_b^c = \frac{c-b}{2(c-a)}$

$==>\ P(R \leq S) = \frac{b-a}{c-a} + \frac{d-c}{d-b} + \frac{c-b}{2(c-a)}$

2. **Case 2: a $\in$ [b, d] et c>d :**

$P(R \leq S) = P(R \leq S,\ a \leq R \leq d,\ a \leq S \leq d)$
$= \int_a^d P(R{=}t,\ S{¿}t)dt = \int_a^d \frac{1}{c-a} \times \frac{d-t}{d-a}dt$
$= \frac{1}{(c-a)(d-a)} \times \left[\frac{-(d-t)^2}{2}\right]_a^d = \frac{d-a}{2(c-a)}$

$==>\ P(R \leq S) = \frac{d-a}{2(c-a)}$

3. **Case 3: [a, c] = [b, d] :**

$P(R \leq S) = \int_a^c P(R{=}t,\ S{¿}t)dt = \int_a^c \frac{1}{c-a} \times \frac{c-t}{d-b}dt$
$= \frac{1}{(c-a)^2} \times \left[\frac{-(c-t)^2}{2}\right]_a^c$
$= \frac{(c-a)^2}{2(c-a)^2} = \frac{1}{2}$

$==>\ P(R \leq S) = \frac{1}{2}$

4. **Case 4: [a, c] $\subset$ [b, d] :**

$P(R \leq S) = P[(c \leq S \leq d) \cup (a \leq R \leq c \cap R \leq S \leq c)$
$= P(c \leq S \leq d) + P(a \leq R \leq c \cap R \leq S \leq c)$

- $P(c \leq S \leq d) = \int_c^d f_S(t)dt = \int_c^d \frac{1}{d-b}dt = \frac{1}{d-b} \times [t]_c^d = \frac{d-c}{d-b}$
- $P(a \leq R \leq c \cap R \leq S \leq c) = P(R \leq S, a \leq R \leq c, a \leq S \leq c)$
$= \int_a^c P(R{=}t, S{¿}t)dt = \int_a^c \frac{1}{c-a} \times \frac{c-t}{c-a}dt$
$= \frac{1}{(c-a)^2} \times \left[\frac{-(c-t)^2}{2}\right]_a^c = \frac{(c-a)^2}{2(c-a)^2}$
$= \frac{1}{2}$

$==>$ $P(R \leq S) = \frac{d-c}{d-b} + \frac{1}{2}$

5. **Case 5: [b, d] $\subset$ [a, c] :**

$P(R \leq S) = P[(a \leq R \leq b) \cup (b \leq R \leq d \cap R \leq S \leq d)$
$= P(a \leq R \leq b) + P(b \leq R \leq d \cap R \leq S \leq d)$
- $P(a \leq R \leq b) = \int_a^b f_R(t)dt = \int_a^b \frac{1}{c-a}dt = \frac{1}{c-a} \times [t]_a^b = \frac{b-a}{c-a}$
- $P(b \leq R \leq d \cap R \leq S \leq d) = P(R \leq S, b \leq R \leq d, b \leq S \leq d)$
$= \int_b^d P(R{=}t, S{¿}t)dt = \int_b^d \frac{1}{c-a} \times \frac{d-t}{d-b}dt$
$= \frac{1}{(c-a)(d-b)} \times \left[\frac{-(d-t)^2}{2}\right]_b^d = \frac{(d-b)^2}{2(c-a)(d-b)}$
$= \frac{d-b}{2(c-a)}$

$==>$ $P(R \leq S) = \frac{b-a}{c-a} + \frac{d-b}{2(c-a)}$

Bibliography

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, (New York, NY, USA), pp. 23–34, ACM, 1979.

[2] S. Christodoulakis, "Implications of certain assumptions in database performance evauation," *ACM Trans. Database Syst.*, vol. 9, pp. 163–186, June 1984.

[3] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, (New York, NY, USA), pp. 268–277, ACM, 1991.

[4] A. Deshpande, M. N. Garofalakis, and R. Rastogi, "Independence is good: Dependency-based histogram synopses for high-dimensional data.," in *SIGMOD Conference* (S. Mehrotra and T. K. Sellis, eds.), pp. 199–210, ACM, 2001.

[5] L. Getoor, B. Taskar, and D. Koller, "Selectivity estimation using probabilistic models," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, (New York, NY, USA), pp. 461–472, ACM, 2001.

[6] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," in *Proceedings of the 1996 ACM SIGMOD*

International Conference on Management of Data, SIGMOD '96, (New York, NY, USA), pp. 294–305, ACM, 1996.

[7] K. Tzoumas, A. Deshpande, and C. S. Jensen, "Lightweight graphical models for selectivity estimation without independence assumptions," *PVLDB*, p. 2011, 2011.

[8] K. Tzoumas, A. Deshpande, and C. S. Jensen, "Efficiently adapting graphical models for selectivity estimation," *The VLDB Journal*, vol. 22, pp. 3–27, Feb. 2013.

[9] C. Moumen, F. Morvan, and A. Hameurlain, "Estimation error-aware query optimization: an overview," *Comput. Syst. Sci. Eng.*, vol. 31, no. 3, 2016.

[10] N. Kabra and D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, (New York, NY, USA), pp. 106–117, ACM, 1998.

[11] N. Bruno, S. Jain, and J. Zhou, "Continuous cloud-scale query optimization and processing," *PVLDB*, vol. 6, no. 11, pp. 961–972, 2013.

[12] T. Neumann and C. A. Galindo-Legaria, "Taking the edge off cardinality estimation errors using incremental execution," in *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pp. 73–92, 2013.

[13] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson, "Dynamically optimizing queries over large scale data platforms," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 943–954, ACM, 2014.

[14] B. Babcock and S. Chaudhuri, "Towards a robust query optimizer: A principled and practical approach," in *Proceedings of the 2005 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '05, (New York, NY, USA), pp. 119–130, ACM, 2005.

[15] F. C. Chu, J. Y. Halpern, and P. Seshadri, "Least expected cost query optimization: An exercise in utility," in *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, pp. 138–147, 1999.

[16] S. Babu, P. Bizarro, and D. DeWitt, "Proactive re-optimization," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, (New York, NY, USA), pp. 107–118, ACM, 2005.

[17] M. Abhirama, S. Bhaumik, A. Dey, H. Shrimal, and J. R. Haritsa, "On the stability of plan costs and the costs of plan stability," *Proc. VLDB Endow.*, vol. 3, pp. 1137–1148, Sept. 2010.

[18] H. D., P. N. Darera, and J. R. Haritsa, "Identifying robust plans through plan diagram reduction," *Proc. VLDB Endow.*, vol. 1, pp. 1124–1140, Aug. 2008.

[19] A. Dutt, S. Neelam, and J. R. Haritsa, "Quest: An exploratory approach to robust query processing," *Proc. VLDB Endow.*, vol. 7, pp. 1585–1588, Aug. 2014.

[20] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, (New York, NY, USA), pp. 659–670, ACM, 2004.

[21] C. Moumen, F. Morvan, and A. Hameurlain, "Handling estimation inaccuracy in query optimization," in *Web Technologies and Applications - 18th Asia-Pacific Web Conference, APWeb 2016, Suzhou, China, September 23-25, 2016. Proceedings, Part II*, pp. 355–367, 2016.

[22] J. M. Schopf and F. Berman, "Using stochastic intervals to predict application behavior on contended resources," 1999.

[23] R. A. T. Joanna M. Papakonstantinou, "Origin and evolution of the secant method in one dimension," *The American Mathematical Monthly*, vol. 120, no. 6, pp. 500–518, 2013.

[24] J. L. Wiener, H. Kuno, and G. Graefe, *Benchmarking Query Execution Robustness*, pp. 153–166. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[25] A. Hulgeri and S. Sudarshan, "Parametric query optimization for linear and piecewise linear cost functions," in *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pp. 167–178, VLDB Endowment, 2002.

[26] R. L. Cole and G. Graefe, "Optimization of dynamic query evaluation plans," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, (New York, NY, USA), pp. 150–160, ACM, 1994.