Robust Query Optimization Methods With Respect to Estimation Errors: A Survey

Shaoyi YIN
IRIT Laboratory
Paul Sabatier University
France
yin@irit.fr

Abdelkader HAMEURLAIN
IRIT Laboratory
Paul Sabatier University
France
hameurlain@irit.fr

Franck MORVAN
IRIT Laboratory
Paul Sabatier University
France
morvan@irit.fr

ABSTRACT
The quality of a query execution plan chosen by a Cost-Based Optimizer (CBO) depends greatly on the estimation accuracy of input parameter values. A slight estimation error may cause significant performance degradation. Plenty of research results have been produced on improving the estimation accuracy, but they do not work for every situation. Therefore, researchers introduced the notion of "robust query optimization", trying to minimize the sub-optimality risk by accepting the fact that estimates could be inaccurate. Dynamic or adaptive query optimization is one way to improve robustness, but not the only way. In this survey, we aim to provide an overview of robust query optimization methods by classifying them into different categories, explaining the essential ideas, listing their advantages and limitations, and comparing them with multiple criteria.

1. INTRODUCTION
Query optimizer is an indispensable component in relational DBMS engines. Since the publication of the famous System-R paper [56], Cost-Based Optimizer is widely adopted. For a given query, the optimizer enumerates all possible execution plans, estimates the cost of each plan using a cost model, and picks the one with least estimated cost. The parameters of the cost model fall into two categories: the database profile and the available amount of system resources. The database profile contains mainly: (1) basic information which represents the properties of the data, such as relation sizes and number of distinct attribute values, and (2) derived information specific to a given query, which is mainly the cardinality (i.e. number of tuples returned by a relational operator). The accuracy of parameter values has a significant impact on the quality of the chosen execution plan.

It has been shown that, even if estimation errors on the basic information are small, their transitive effect on estimates of the derived information can be devastating (e.g. the error propagates exponentially with respect to the number of joins) [38]. Consequently, the optimizer may choose a wrong plan. However, due to non-uniform distribution of attribute values and correlations between attributes, the cardinality estimation problem remains very challenging. A lot of efforts have been made to improve the estimation accuracy, such as histograms [40], sampling-based methods [44,45,52], maximum entropy-based methods [47] and probabilistic graphical model-based methods [60,61]. Nevertheless, they do not work well for every situation, in particular for complex predicates and skewed data. In addition, the available amount of system resources may change dynamically during query execution.

Although having accepted the fact that parameter value estimates could be inaccurate or even missing, researchers still desire to minimize the plan sub-optimality risk, so they introduced the notion of "robustness" to query optimizer. Informally, robustness means the ability of error resistance. However, there is not yet a formal definition of robustness for query optimization recognized by everyone. Recently, Graefe et al. have organized two seminars [27,29] and one research panel [28] on the "robust query processing" topic. Before that, they tried to visualize and benchmark the robustness of query execution [26, 63]. [63] distinguished three types of robustness: query optimizer robustness ("the ability of the optimizer to choose a good plan as expected conditions change"), query execution robustness ("the ability of the query execution engine to process a given plan efficiently under different runtime conditions") and workload management robustness (which "characterizes how vulnerable database system performance is to unexpected query performance"). Each type deserves an in-depth study. In this survey, we focus on the first one: query optimizer robustness. To make the concept more concrete, we propose the following definition: a query optimizer is robust with respect to estimation errors, if it is able to find a plan (or several plans) such that the query response time T is not greater than $(1 + \varepsilon) * T(P_{opt})$ despite of estimation errors, where $T(P_{opt})$ is the query response time by executing the optimal plan P_{opt} implied by exact input parameter values and ε is a user-defined tolerance threshold. Note that, how to obtain efficiently all the exact parameter values to find P_{opt} (the baseline for comparison) is also challenging [17], but it is outside the scope of our survey.

The above statement is the main objective for a robust query optimizer. Although it has not yet been achieved completely, some "best-effort" research results are worth being studied. Some of them have been analyzed in previous surveys under the terms like "dynamic query optimization" [19,48] or "adaptive query processing" [6,21,30,31]. Indeed, dynamic or adaptive query optimization is one way to improve robustness. However, as you will see, there are also other interesting approaches proposed for this purpose. The objective of our survey is to give an overview of the representative robust query optimization methods, including many recent proposals [13,14,15,22,24,35, 36,49,50,51,59] that have not yet been covered in any survey. The major contributions are: (1) proposing a two-level classification framework for robust query optimization methods, (2) illustrating the inherent advantages and limitations of each method, as well as the relationship between them, and (3) comparing the methods using multiple well-chosen criteria.

The remainder of the paper is organized as follows. In Section 2, we describe the proposed classification framework and choose multiple criteria for later comparisons. Section 3 and Section 4 are organized in accordance with the classification framework. We analyze the representative methods and compare them using the chosen criteria. In Section 5, we give a global comparison of the main approaches and their sub-classes. Finally, we conclude the paper in Section 6.

2. CLASSIFICATION AND CRITERIA

2.1 Classification Framework

According to the output of the query optimizer (for a given query), we distinguish the following two approaches: (1) single-plan based approach, where the output of the optimizer is a single plan with the least estimated cost, and (2) multi-plan based approach, where the output of the optimizer is a set of plans. The main difference between them is that the latter often requires a more sophisticated execution model.

For the single-plan based approach, we categorize the methods into three classes: **(1) Cardinality Injection (CI)**. Instead of deriving cardinalities from basic database statistics, these methods try to obtain directly the cardinality values for some operators. The main objective is to overcome the correlation problem (w.r.t. multi-predicate selections) and limit the error propagation effect (w.r.t. joins). One way is to collect information from execution feedback of previous queries; another way is to execute some important sub-queries during optimization. **(2) Plan Modification (PM)**. These methods collect statistics and detect estimation errors during query execution, then react to them by modifying the plan dynamically. Sometimes, they may need to recall the optimizer repeatedly. **(3) "Robust Plan" Selection (RPS)**. In this class, the optimizer does not choose an "optimal" plan, but a "robust" plan, i.e. a plan which is less sensitive to estimation errors. The main difference between the above classes lies in the way that the query optimizer and the plan executor interleave. With CI, for a given query, the optimizer runs only once, but it consumes the feedback of the executor; with PM, for a given query, the optimizer may run several times during the query execution; with RPS, for a given query, the optimizer is expected to run only once, and there is no interleaving between the optimizer and the executor.

For the multi-plan based approach, we also categorize the methods into three classes: **(1) Deferred Plan Choosing (DPC)**. The optimizer proposes several potentially optimal plans, and the final choice is done during execution time. One way is to run these plans in a competition mode. Another way is to start with one plan and smoothly switch to others if necessary. **(2) Tuple Routing (TR)**. Avnur et al. proposed a special operator called "eddy" [5] which receives all base relation tuples and intermediate result tuples, then routes them through the relational operators in different orders. Since different tuples may follow different routing orders, and each routing order corresponds to a specific execution plan, we consider this mechanism belonging to the multi-plan based approach. **(3) Data Partitioning (DP)**. In this class, the optimizer partitions the data explicitly according to their inherent characteristics such as skewed distribution or correlations, such that different optimal plans may be executed for different data partitions. The main difference between these classes lies in how to decide which plan is used for which data. With DPC, only one plan will be finally chosen and used for the complete dataset; with TR, the eddy operator chooses a routing order (i.e. a plan) for each tuple, and the decision is based on local statistics collected by the eddy; with DP, the optimizer decides the mapping between sub-datasets and multiple plans, based on global statistics.

A summary of the classification is shown in Figure 1.

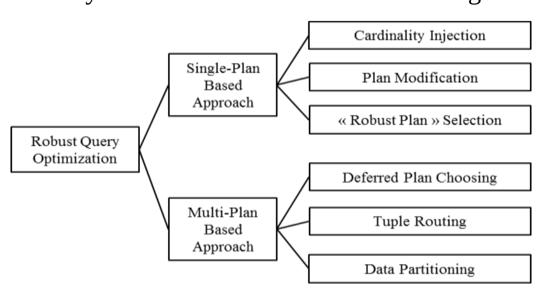


Figure 1. Classification of robust optimization methods

2.2 Comparison Criteria

For each approach, different methods will be compared using the following criteria. The first three criteria define the application scope; the fourth is related to query performance; and the last concerns the software engineering aspect. Here, we list the options for each criterion and their abbreviations. The abbreviations will be used later in comparison tables.

C1: Estimation Error Sources. The existing methods deal with one or several of the following estimation error sources: non-uniform data distribution (DD), data correlation (DC), statistics obsoleteness due to data modification (DM), missing statistics (MS, e.g. for complex predicates or remote data sources), unavailability of resources (UR), data arrival delay (AD), data arrival rate changing (AR), and so on.

C2: Target Query Types. Some methods aim at optimizing the currently running query (C); some serve for future executions (F) of the same (Sa) or similar (Si) queries; others only deal with predefined parameterized queries (P).

C3: Target Optimization Decisions. Due to estimation errors, the optimizer may make wrong decisions in the following aspects: base relation access methods (AM), join methods (JM), join order (JO), operator execution order (OEO), execution site (ES) of an operator, CPU allocation (CA), memory allocation (MA), parallelism degree (PD), partitioning key (PK), etc. Different methods may cover different aspects.

C4: Performance Degradation Risk. Due to the coexistence of numerous uncertain factors, most of the existing methods cannot yet provide robustness guarantees. Sometimes, for a given query, the "robustly" chosen plan becomes even worse than the "naively" chosen plan (i.e. the plan chosen by a classical optimizer). To compare the methods, we say that: the risk is high (H) if there is no worst-case guarantee; the risk is low (L) if the degradation is constant or linear to the original performance; the risk is medium (M) if the maximum degradation is bounded, but non-linear to the original performance. Sometimes, the risk level is low but only under certain conditions (LC). It is also possible that there is no degradation risk (N) or that the user is allowed to choose a certain risk level (UC).

C5: Engineering Cost. After years' of maintenance, the commercial DBMS engines become extremely complex. Modifications should be made very carefully to avoid system regression. We assess the engineering cost like this: low (L), if only adding a stand-alone module which could be turned on/off easily; medium (M), if just modifying a few modules of the optimizer or the executor; or high (H), if rewriting most of the optimizer or the executor code.

3. SINGLE-PLAN BASED APPROACH

3.1 Cardinality Injection

"Cardinality Injection" is a concept introduced by Chaudhuri in [16], meaning that the cardinality information is obtained from other sources rather than derived from basic database statistics. To obtain the cardinality, there exist mainly two different ways: learning from previous query executions and running some sub-queries during optimization process.

3.1.1 Exploiting the execution feedback

LEO (DB2's LEarning Optimizer) [57] is the most representative work of using query execution feedback for cardinality estimation. LEO captures the number of rows produced by each operator at run-time, by adding a counter in the operator implementation. Then, the <predicate, cardinality> pairs are stored in a feedback cache, which can be consulted by the query optimizer in conjunction with catalog statistics when optimizing a future query.

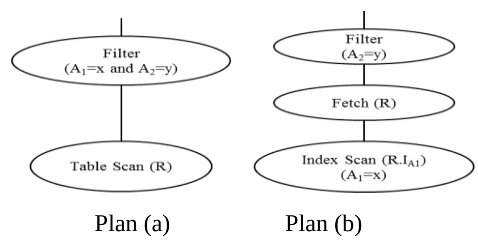


Figure 2. Example to show the limitation of LEO

However, using this mechanism, we can only obtain the cardinalities of a subset of predicates used by the optimizer to estimate the costs. For example, given the query "select * from R where A_1=x and A_2=y", if the executed plan is Plan (a) in Figure 2, we can obtain the cardinality Card(A_1=x and A_2=y) from the first execution, so in the future, the cost estimation for Plan (a) will be more accurate. However, to estimate the cost of Plan (b), the optimizer still need to derive the cardinality Card(A_1=x) from catalog statistics. To solve this problem, the pay-as-you-go (PAYG) framework [15] uses proactive plan modification and monitoring mechanisms, in order to obtain cardinality values for some sub-expressions (e.g. Card(A_1=x)) which are not included in the running plan. For example, when running Plan (a), in addition to counting the number of rows satisfying the predicate "A_1=x and A_2=y", the operator keeps another counter which is increased every time when "A_1=x" is true for an input tuple. Thus, in the future, the optimizer can also estimate the cost of Plan (b) more precisely. Obviously, this will increase the overhead of query execution, however, the DBA or the user is allowed to specify a bound on the additional overhead for a query.

Some other methods [1,18] use feedback information to refine the catalog statistics. They are less flexible, because they limit to a certain statistical data structure, while cardinality injection methods are independent of the underlying representation of database statistics.

3.1.2 Querying during optimization process

This idea was first adopted for multi-databases (MIND system [23]). In case that there are not enough statistics for generating a complete execution plan, the query optimizer first decomposes the query into sub-queries, sends them to different remote sites, and then decides the order of inter-site operators (e.g. joins and unions) based on the sub-query results. A recent work of Neumann et al called "Incremental Execution Framework" (IEF) [49] adopted a similar principle to optimize queries in uni-processor environment. The main steps are: (1) construct the optimal execution plan using the standard cost model, (2) identify sensitive plan fragments, i.e. the fragments whose cardinality estimation errors might lead to wrong plan decisions for the higher level operators, (3) execute those plan fragments, materialize the results, and thus retrieve the actual cardinality (i.e. the number of tuples produced by an operator at run-time), and (4) find a new optimal plan using the obtained cardinality. IEF to some extent removes the uncertainty of the cardinality estimation. However, it still has some limitations. For example, it identifies the sensitive query fragments based on "estimation error rates", but this information is often unknown or inaccurate.

Different from the above work, Xplus [36] focused on offline tuning of repeatedly-running queries. When a query plan is claimed to be sub-optimal, the optimizer picks some candidate (sub)plans heuristically, calls the executor to run these (sub)plans iteratively, collects the actual cardinalities from these runs, and stops the iterations when finding a plan which is δ% better than the original one, where δ is a user-defined threshold. This proposal is only suitable for read-only applications (e.g. OLAP) running in a stable environment.

3.1.3 Comparison

In Table 1, we compare the above methods using the five criteria defined in Section 2.2. C1: They deal with all kinds of cardinality estimation errors, except that Xplus is not resistant to data modifications. C2: The feedback-based methods serve for the future similar queries; others optimize the currently running query, except that Xplus tunes a query for future runs. C3: LEO, PAYG and Xplus try to improve all kinds of decisions; IEF focuses on JM and JO. C4: As LEO and PAYG only correct part of the estimation errors, the uncertainty of other values may lead to a worse plan. However, for repeatedly-running queries, after several runs, a stable plan can be obtained. Normally, PAYG could converge earlier, because it collects information more efficiently. IEF introduces some degradation risk due to materialization, but this cost is rather limited. Xplus does not have degradation risk, because it uses only exact cardinalities. C5: LEO is comprised of four components: a component to save the optimizer's plan, a monitoring component, a feedback analysis component and a feedback exploitation component. The analysis component is a stand-alone process, and the others are minor modifications to the DB2 server. PAYG needs to modify the optimizer generated plan and identify important expressions to run, so it is more complicated than LEO. IEF optimizer needs to identify critical plan fragments to execute, interact with the executor, and materialize intermediate results, but the plan executor keeps the same, so we consider the engineering cost as medium. Finally, Xplus can work as a stand-alone module.

Table 1. Comparison of cardinality injection methods

Criterion / Method	**C1: Estim. error sources**	**C2: Target query types**	**C3:Target opt. decisions**	**C4:Deg. risk**	**C5:Eng. cost**
LEO	DD,DC,DM,MS	F, Si	All	LC	L
PAYG	DD,DC,DM,MS	F, Si	All	LC	M
IEF	DD,DC,DM,MS	C	JM,JO	L	M
Xplus	DD,DC,MS	F, Sa	All	N	L

3.2 Plan Modification

Differently, in this class, the optimizer uses catalog statistics to generate a plan. However, the execution plan is monitored at run-time. Once a sub-optimality is detected, the plan is modified: either by rescheduling, or by re-optimization. Rescheduling is to update only the execution order of the operators or to update the order of the base relations. Re-optimization is to generate a new plan for the remainder of the query using run-time collected statistics.

3.2.1 Rescheduling

In distributed environment, the relations participating in a query plan are often stored in remote sites, and the arrival of data may be delayed. In this situation, to avoid idling, Amsaleg et al proposed a query plan scrambling algorithm [2]. The algorithm contains two phases: (1) materializing sub-trees. During this phase, each iteration of the algorithm identifies a plan fragment that is not dependent on any delayed data (the fragment is called a "runnable sub-tree"), then the fragment is executed and the result is materialized. (2) Creating new joins between relations that were not directly joined in the original query tree. When no more runnable sub-trees can be found by Phase 1, the scrambling algorithm moves into Phase 2, so that the plan execution could be continued.

Query plan scrambling can improve the response time in many cases, but it only deals with the initial delay (i.e. the arrival of the first tuple is delayed). If the delay happens during the execution of the fragment, it is blocked and has to wait. To solve this problem, Bouganim et al proposed a dynamic query scheduling strategy (DQS) [12] that interleaves the scheduling phase and the execution phase. Each time, the scheduler only schedules the query fragments that can be executed immediately (i.e. all their inputs are available and there is enough memory). These fragments are executed concurrently. During execution, the data arrival rate is monitored continuously. Once a problem is detected or the execution is finished, rescheduling is triggered. If the rescheduling cannot solve the problem, a re-optimization (see Section 3.2.2) may be triggered.

3.2.2 Re-optimization

The dynamic Re-Optimization (ReOpt) algorithm proposed by Kabra et al [43] detects the estimation errors during query execution and re-optimizes the rest of the query if necessary. At specific intermediate points in the query plan, statistics collector operators are inserted to collect various statistics, such as cardinalities, minimum and maximum values for an attribute, and even histograms. During query execution, the collected statistics are compared with the estimated ones. If there is a big difference, some heuristics are triggered to evaluate whether a re-optimization is beneficial. If so, the optimizer is recalled to modify the execution plan for the remainder of the query. Instead of suspending a query in mid-execution (which is difficult to implement), they let the currently executing operator run to completion and re-direct the output to a temporary file on disk. Then, SQL corresponding to the remainder of the query is generated by using this temporary file. The new SQL is re-submitted to the optimizer as a regular query. In ReOpt, if the difference between the collected parameter value and the estimated one exceeds a threshold, the re-optimization procedure will be considered. A later work [46] argued that this threshold is chosen arbitrarily so could be blind. For example, in Figure 3, for query Q1, the estimated cardinality is ec1, so the initially chosen plan is P1a according to the cost function; assuming that the actual cardinality is ac1 = 1.5 * ec1, so P1a is actually sub-optimal. For query Q2, the estimated cardinality is ec2, so the initially chosen plan is P2b; assuming that the actual cardinality is ac2 = 1.5 * ec2, so P2b is indeed optimal. Suppose that the threshold used by the ReOpt algorithm is less than 0.5, then both queries may be re-optimized; otherwise, if the threshold is greater than 0.5, then neither of the two queries will be re-optimized. However, the expected reaction is that Q1 is re-optimized, while Q2 is not. In order to have such a reaction, [46] introduced POP algorithm which uses the concept "validity range" of the chosen plan for each input parameter. If the actual value of the parameter violates the validity range, a re-optimization is triggered; otherwise, the current plan continues executing. For example, the validity range of P1a for the given parameter is [0, c], and the validity range of P2b is [c, ∞). Obviously, only Q1 will be re-optimized. The violation of validity ranges is detected by a CHECK operator. Han et al [33] extended the POP algorithm to a parallel shared-nothing environment.

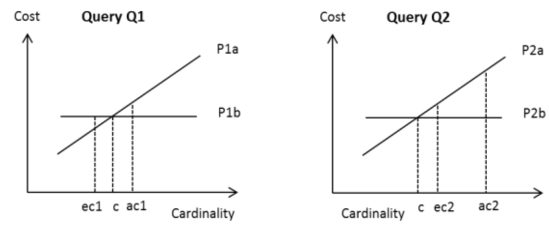


Figure 3. Introduction of the validity range

Continuous query optimization (CQO) [13] extends ReOpt into the context of cloud-scale query optimization, where massively parallel processing is a natural choice. On the same principle, they continuously monitor query execution, collect runtime statistics and dynamically modify parallelism degree or the partition key choice. Similarly, ReOpt is also applied for recursive queries in [24]. Bonneau et al [11] and Hameurlain et al [32] focused on the re-optimization problem for the parallel shared-nothing architecture and the multi-user environment. The main objective is to improve the physical resource (CPU and memory) allocation by exploiting the collected statistics at run-time. When an estimation error is detected, they re-optimize not only the mapping between the remaining tasks and the CPUs, but also the allocation of memory. Incremental memory allocation (we call it IMA in short) heuristics were proposed to avoid unexpected extra I/Os caused by lack of memory. During re-optimization, the parallelism degree may also be modified to satisfy the memory requirements.

3.2.3 Comparison

In Table 2, we compare the above methods using the five criteria. C1: The rescheduling methods were mainly designed to deal with data arrival delay and rate changing problems, while the re-optimization methods were originally used to solve the other estimation error problems. However, they are not contradictory and can co-exist to handle both kinds of problems, as Tukwila [41] does. In CQO, statistics are missing during optimization. In IMA, the unavailability of resources is also taken into account. C2: They all optimize the currently running query. C3: They focus on different decision aspects, but none of them deal with all aspects. C4: Scrambling may degrade the performance dramatically if a bad join order is chosen during Phase 2. DQS reduces this risk by estimating the increased cost before changing the join order. ReOpt reacts to every detected estimation error. When there is only one wrongly estimated parameter, it works very well. However, when there are many uncertain parameters, each re-optimization may generate just another wrong plan. POP and IMA may face the same problem. We consider the degradation risk level as high. CQO is different, because for the moment, it only modifies parallelism degree and partition key. The decision is based on run-time collected statistics, so the degradation risk is rather low. C5: Scrambling only modifies slightly the scheduler in order to detect data arrival delays and run the two phases iteratively. DQS not only rewrites the scheduler, but also modifies slightly the optimizer to generate annotated query plans and enhances the executor to be able to interact with the scheduler. Re-optimization methods only add statistics collectors and re-optimization triggers, except POP. It suspends the query during re-optimization, and it provides different strategies for placing the CHECK operators, in order to support pipelined execution.

Table 2. Comparison of plan modification methods

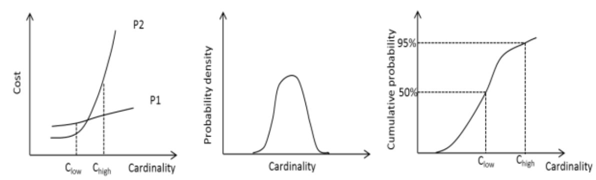
Criterion / Method	C1: Estim. error sources	C2:Target query types	C3:Target Opt. decisions	C4:Deg. risk	C5:Eng. cost
Scrambling	AD	C	OEO,JO	H	L
DQS	AD,AR	C	OEO,JO	L	M
ReOpt	DD,DC,DM	C	JM,JO,MA	H	L
POP	DD,DC,DM	C	AM,JM,JO	H	M
CQO	MS	C	PD,PK	L	L
IMA	DD,DC,DM,UR	C	CA,MA,PD	H	L

3.3 "Robust Plan" Selection

In section 3.2, we presented the methods which react to estimation errors by modifying the plan; in this section, we examine the methods which take into account the uncertainty during optimization time. Instead of an "optimal" plan, they choose a "robust" plan.

3.3.1 *Robust cardinality estimation*

The traditional sampling-based cardinality estimation methods compute a single-point value: if the population size is N, the sample size is s, and the observed cardinality for the sample is C', then the estimated cardinality C for the whole dataset should be C'*N/s. The estimation error e could be very small; however, the optimizer may still make a big mistake when choosing the plan. For example, in Figure 4 (a), we have two candidate plans P1 and P2 for query Q, where x-axis represents the value space of the uncertain cardinality and y-axis represents the cost of the plan. Suppose that the real cardinality is between C_{low} and C_{high}. If the estimated cardinality is C_{low}, the optimizer will choose P2. Otherwise, if the estimated cardinality is C_{high}, the optimizer will choose P1. By comparing these two situations, we find that the first situation is more risky, because the worst case cost is very high.



(a) (b) (c)

Figure 4. Robust cardinality estimation

Instead of estimating the cardinality by a single value C, Robust Cardinality Estimation (RCE) method in [8] derives a probability density function of C from the sampling result, as shown in Figure 4(b). Then it transforms the probability density function to a cumulative probability function *cdf(c)*, as shown in Figure 4(c). The user is allowed to choose a confidence threshold *T* which represents the level of risk (a big *T* corresponds to a small risk). The estimated cardinality is computed by: $C = cdf^{-1}(T)$. For example, if *T*=95%, the model returns C_{high} as the estimated cardinality, so the more stable plan P1 will be chosen. If *T*=50%, the more risky plan P2 will be chosen. They say that this solution is robust, because users are aware of the risk and take the responsibility for it.

3.3.2 *Proactive re-optimization*

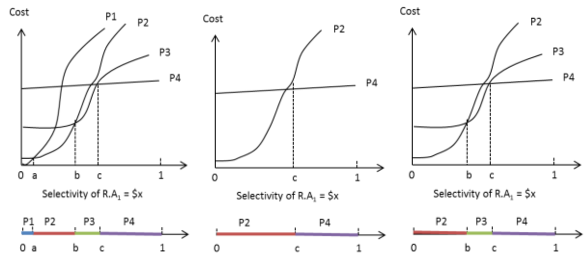
Babu et al proposed another way to take into account the estimation uncertainty during optimization process. In the Rio prototype [7], they estimate cardinalities using intervals, instead of single point values. If the optimizer is very certain of the estimate, then the interval should be narrow; otherwise, the interval should be wider. Using intervals allows the optimizer to generate robust plans that minimize the need for re-optimization. A robust plan is a plan whose cost is very close to optimal at all points within the interval. For example, in Figure 4(a), we assume that the estimated cardinality is C_{low} and the actual cardinality is C_{high}. With re-optimization methods such as POP, the optimizer will first choose P2 which is optimal at point C_{low}, and then choose P1 during re-optimization. For more complicated queries (e.g. with multiple joins and selection predicates), the situation could be even worse: re-optimization may happen repeatedly when multiple errors are detected.

We come back to the example in Figure 4(a). With Rio, the cardinality could be estimated as interval [C_{low}, C_{high}], so P1 will be chosen directly at the beginning, because it is robust within this interval. Thus, the re-optimization is avoided. They say that their method is "proactive", because instead of reacting to the disaster caused by a wrong plan, they tried to prevent the optimizer from choosing that plan. Unfortunately, very often, a robust plan does not exist for the estimated interval. In this case, the authors propose to choose a set of plans which are "switchable". We will talk about this in Section 4.1. Actually, sometimes, we cannot even find a switchable plan. If this is the case, the authors propose to do like POP: choose an optimal plan using a single-point estimate and re-optimize the query if necessary. Note that, even a robust or switchable plan is found, re-optimization may still be triggered, because the actual cardinalities may be outside of the estimated intervals.

Ergenç et al [22] extends the proactive re-optimization idea to deal with query optimization problem in large scale distributed environments. In such environments, the amount of data transferred between sites has a big impact on the overall performance. If the optimizer decides to place a relational operator at a wrong site due to cardinality estimation errors, huge amount of data may be transferred. To minimize the risk of wrong placement, they estimate the cardinality as an interval instead of a single point value. If at any point in the interval, placing an operator on site S provides near-optimality (i.e. the performance degradation compared to the optimal placement is less than a threshold), then the site S is called a robust site. A Robust Placement (RP) for a query is to place recursively each operator in the plan tree on a robust site.

3.3.3 Robust plan diagram reduction

A "plan diagram" is a color-coded pictorial enumeration of the plans chosen by the optimizer for a parameterized query template over the relational selectivity space [55]. The diagram is generated offline by a tool called Picasso [53] by repeatedly invoking the query optimizer, each time with a different selectivity value. Then, the diagram is used by the query optimizer as follows: for an instance of the query template, the optimizer first calls the selectivity estimator, and then picks the corresponding plan from the diagram. In the original paper, they gave examples with two-dimensional diagrams, each dimension representing the possible selectivity of one parameterized predicate in the query template. In this paper, for ease of comprehension, we illustrate the principle through a one-dimensional example. The example query is "select * from R, S where $R.A_2 = S.A_2$ and $R.A_1$ = \$x", where \$x is a variable. Figure 5(a) shows the diagram in the bottom, and the corresponding cost function curves above for more information. For example, if the estimated selectivity is between b and c, plan P3 will be chosen.



(a) (b) (c)

Figure 5. Example of robust plan diagram reduction

Using the plan diagram can avoid running the complete optimization algorithm for each instance of the parameterized query, thus the query optimization is more efficient. However, a plan diagram may contain a large number (e.g. more than 100) of plans in the selectivity space, making the diagram maintenance difficult. Therefore, Harish et al [34] proposed to reduce the dense diagrams to simpler ones, without degrading too much the quality of each individual plan. The principle is as follows: a plan Pa can be replaced by another plan Pb if and only if at each query point covered by Pa, the increased cost *(C(Pb) – C(Pa))* is less than a tolerance threshold defined by the user (such as 10%). For example, P1 and P3 can be replaced by P2, as shown in Figure 5(b). When the cardinality estimation is precise, the reduced plan diagram does not degrade the performance too much compared to the original diagram. However, when there are estimation errors, there could be a high risk to use the reduced diagram. For example, with the reduced diagram in Figure 5(b), if the estimated selectivity is between b and c, P2 will be chosen. However, if the actual selectivity is much higher than c, the cost of P2 becomes extremely high compared to P3.

To reduce this risk, the same authors [35] proposed to make the plan replacement policy stricter: during the plan reduction, a plan Pa can be replace by another plan Pb only if at each query point in the **whole selectivity space**, the increased cost *(C(Pb) – C(Pa))* is less than a tolerance threshold defined by the user. Considering this condition, P3 cannot be replaced anymore, so we get the reduction result in Figure 5(c). This reduction is called "Robust Diagram Reduction" (RDR), because the risk of significant performance degradation is limited in case of estimation errors.

3.3.4 Comparison

The above methods are compared in Table 3 using the five criteria. C1: They all take into consideration the uncertainty of parameter values at compile time. Although not pointed explicitly, they have made the following assumptions: catalog statistics or samples are maintained up to date by the system; the running environment is stable. C2: They all deal with the currently running query, except RDR, which works for predefined parameterized queries. C3: RCE, Rio, RDR could avoid wrong decisions on base relation access methods, join methods and join ordering; RP extends Rio to improve the execution site selection.

Table 3. Comparison of robust plan selection methods

Criterion / Method	C1: Estim. error sources	C2: Target query types	C3: Target opt. decisions	C4:Deg. risk	C5:Eng. cost
RCE	DD,DC	C	AM,JM,JO	UC	L
Rio	DD,DC	C	AM,JM,JO	LC	M
RP	DD,DC	C	AM,JM,JO,ES	LC	M
RDR	DD,DC	P	AM,JM,JO	L	L

C4: RCE allows the user to choose the risk level. For Rio, if the actual cardinalities fall into the estimated intervals, and if a robust plan exists, then the performance degradation risk is limited to a predefined threshold. However, these two conditions are difficult to satisfy. RP has the same risk level as Rio. The degradation risk of RDR is always limited, thanks to the strict replacement policy. C5: RCE only modifies the cardinality estimation module of the optimizer. Rio requires more modifications to the DBMS engine. RP works for large-scale distributed environments and is based on a mobile execution model [4]. Similar to Rio, it also requires significant modifications to the optimizer in order to support the interval-based estimation. RDR develops a stand-alone tool to prepare a set of robust plans for a predefined query template.

4. MULTI-PLAN BASED APPROACH

4.1 Deferred Plan Choosing

Parametric query optimization [39,10] is used to compile parameterized queries. The optimizer determines a set of plans during optimization, such that for each point in the parameter space, the optimal plan is included in the set. It defers the plan choosing until the beginning of execution. However, its objective is just to avoid compiling the query for each run, but not to achieve robustness. Indeed, it does not deal with cardinality estimation errors or run-time environment changing. In this section, we will study other methods, which make the choice in the middle of execution.

4.1.1 Access method competition

Whether to use indexes and which indexes to use for a single-relation access depends strongly on the selectivity of the predicate. To avoid wrong optimization decisions due to the selectivity estimation uncertainty, Antoshenkov [3] proposed access method competition (AMC), i.e. to run simultaneously different base relation access processes for a small amount of time. They argue that there is a high probability that one of them finishes during this time, and others can be canceled. Otherwise, if none of them finishes quickly, the execution engine should guess and continue only one that has the least estimated cost.

4.1.2 Plan switching

In Rio [7], if the optimizer fails to find a robust plan within the estimated interval, it tries to find a "switchable plan" (SP), which is a set S of plans such that: (1) at any point in the intervals, there is a plan p in S whose cost is close to optimal; (2) according to the detected statistics, the system can switch from one plan to another in S without losing any significant fraction of work done so far. Figure 6 gives an example of a switchable plan.

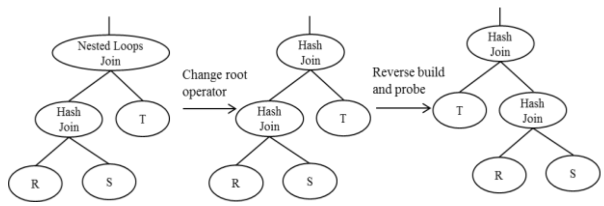


Figure 6. Example of a switchable plan

Assuming that the result size of R∞S is estimated to be very small, so the first plan is executed at the beginning; during execution, if the tuples produced by R∞S cannot fit in memory, the second plan will be switched on (i.e. changing the join algorithm from NLJ to HJ); later on, if the result size of R∞S is detected to be much bigger than the relation T, the third plan will be switched on (i.e. reversing build and probe). The switching process is done smoothly thanks to a "switch" operator integrated in the plan tree.

The idea is similar to the "choose-plan" operator in [25]. However, the latter was originally designed to avoid recompiling every instance of a predefined parameterized query, and it does not deal with cardinality estimation errors caused by non-uniform data distribution or data correlation. Thus, we do not include it specifically as a robust optimization method.

4.1.3 Comparison

In Table 4, we compare the above two methods using the five criteria. C1: SP computes estimation intervals based on catalog statistics, so if data are modified frequently or no statistics are available, the estimated intervals could be erroneous. On the contrary, AMC does not rely on estimations, so it is resistant to data modification and missing statistics. C2: Both methods work for the current query. C3: AMC focuses on base relation access method selection; SP deals with join methods and join order. C4: AMC has low degradation risk, at the condition that one of the concurrent processes finishes quickly; SP also has low degradation risk, at the condition that the actual cardinalities are inside the estimated intervals. C5: Both methods require major modifications to the optimizer and the executor.

Table 4. Comp. of deferred plan choosing methods

Criterion / Method	C1: Estim. error sources	C2: Target query types	C3:Target opt. decisions	C4:Deg. risk	C5:Eng. cost
AMC	DD,DC,DM,MS	C	AM	LC	M
SP	DD,DC	C	JM,JO	LC	M

4.2 Tuple Routing

With the methods in Section 4.1, although the optimizer proposes multiple execution plans, only one of them "survives". Thus, all tuples flow through the same plan tree (route). Differently, Avnur et al [5] allow different tuples to flow through different routes using a special operator "eddy". The eddy mechanism was extended for different environments [54,58,64].

4.2.1 The eddy

A tree-like plan fixes the execution order of the operators in advance, i.e. tuples always flow from leaves to the root. This order can be changed during query execution by using rescheduling or re-optimization, but it is too costly to change it frequently. Avnur et al. [5] proposed a more flexible mechanism which can continuously reorder the operators. They use a star-like query plan, where the relational operators surround a coordinating operator called an eddy. Tuples from base relations and intermediate results are first sent to the eddy. The eddy then routes each tuple to an operator according to the routing policy. Only if a tuple has been handled by all the operators, the eddy sends it to the output.

We illustrate the effectiveness of eddies using the query "select * from R, S, T where R.B=S.B and S.C=T.C". Suppose that tuples from the three relations arrive at different delay and different rate. In the plan, if there are two join operators Op1 ($R\infty S$) and Op2 ($S\infty T$), tuples from R can be only routed to Op1, and tuples from T can be only routed to Op2, while tuples from S can be routed either to Op1 or to Op2. The eddy makes the routing decision for tuples from S dynamically according to a predefined policy, which tries to minimize the execution cost. A specific join algorithm called Symmetric Hash Join (SHJ) [62,37] is recommended: two hash tables are maintained, one for each input relation; the arriving tuple is built immediately into the corresponding hash table and probed against the existing tuples in the other hash table, so the intermediate result can be produced immediately and returned to the eddy. This is actually the "secret" of eddies to enable the reorder-ability: (1) the operator state is continuously maintained, regardless of the execution order; and (2) the "faster" relation is never blocked by the "slower" relation.

We can find that, an eddy is equivalent to a set of tree-like query plans, each one handling a subset of tuples. The tuple routing policy is used to make the mappings. Advanced routing policies [9] were proposed later on.

4.2.2 Extensions of the eddy

The adaptability of the eddy is limited to operator re-ordering, whereas access methods and join algorithms are pre-chosen and fixed during the execution. A more flexible version [54] allows also continuously changing the choice of access methods and join algorithms, etc. To do this, Raman et al [54] made the following main modifications to the eddy architecture: (1) each join operator is replaced by two State Modules (SteMs) which encapsulate data structures (such as hash tables or indexes) used in traditional join algorithms; (2) one or several Access Modules (AMs) are added to each base relation, each AM encapsulating one access method to the data source; and (3) new routing policies are used by the eddy module. At the beginning, different AMs are run concurrently (thus redundantly). In fact, they are in competition: when the eddy finds that one is much more efficient than the others, it will stop the slower ones. Tian and Dewitt [58] extended the eddy and SteMs architecture to a distributed version. Instead of using a centralized eddy module which could become a bottleneck, they integrate the routing function into each single operator. Zhou et al [64] designed another distributed query processing architecture called SwAP, building on eddies and SteMs. They use one eddy module for each execution site, which routes tuples between the local operators and remote eddies.

4.2.3 Comparison

We compare the tuple routing methods in Table 5 using the five criteria. C1: They deal with all kinds of estimation errors. C2: They optimize the currently running query. C3: Eddy only optimizes the join order and operator execution order; the extensions of eddy also optimize base relation access methods and join methods. C4: Although there is no theoretical guarantee, Deshpande [20] has shown experimentally that the performance degradation of eddies is low. Other methods have the same risk level, under the condition that one of the competing access methods wins quickly. C5: In these methods, most of the optimization decisions are made by the eddy module, so the classical optimizer is reduced to a pre-optimizer, and the execution engine becomes more complicated. Thus, the engineering cost is high.

Table 5. Comparison of tuple routing methods

Criterion / Method	C1: Estim. error sources	C2: Target query types	C3: Target opt. decisions	C4:Deg. risk	C5:Eng. cost
Eddy	All	C	JO,OEO	L	H
SteM	All	C	AM,JM,JO,OEO	LC	H
Tian	All	C	AM,JM,JO,OEO	LC	H
SwAP	All	C	AM,JM,JO,OEO	LC	H

4.3 Data Partitioning

In tuple routing based methods, the mapping between tuples and multiple plans is decided by the eddy module, according to local indicators, such as input rate and output rate of an operator. In data partitioning based methods, the mapping is decided by the optimizer, according to global statistics, such as data distribution characteristics and detected correlations between attributes. Tuple routing tends to avoid worst-case performance, while data partitioning also aims at exploring best-case opportunities. We will present some representative methods in this section.

4.3.1 Run-time partitioning

Ives et al [42] proposed adaptive data partitioning (ADP) method. During query execution, the data are dynamically partitioned into sub-datasets, each following a specific plan. Three partitioning strategies were illustrated: (1) sequential partitioning. The query execution is divided into multiple phases. All tuples arriving during Phase Ph_N follow a plan Pl_N. Pl_N is chosen using the statistics collected during the previous N-1 phases. To guarantee the correctness of the result, a "stitch-up" phase is added in the end. For example, two relations S and T are joined through two phases. During Ph_0, S_0 and T_0 are joined; during Ph_1, S_1 and T_1 are joined. According to the following equation: $S \infty T = (S_0 \infty T_0) \cup (S_1 \infty T_1) \cup (S_0 \infty T_1) \cup (S_1 \infty T_0)$, the "stitch-up" phase has to compute $(S_0 \infty T_1)$ and $(S_1 \infty T_0)$. (2) Dynamic splitting. Multiple plans are run concurrently, and the arriving tuple is sent to one plan by a "split" operator according to some criteria. For example, to join two relations which are quasi-sorted, tuples respecting the expected order will be sent to the plan with merge-join, and others will be sent to the plan with hash join. (3) Partitioning used for plan competition. Multiple plans are run concurrently, each processing a small subset of data. If one is much faster than the others, it will process all the remaining data. Note that, for the last two strategies, a "stitch-up" phase is also needed.

4.3.2 Compile-time partitioning

Different from ADP, Query Mesh (QM) model [50,51] decides the data partitions and corresponding plans at query compile time. For a given query, a decision tree-based classifier is learned from a training dataset. Each decision node is a predicate (such as A>x) which distributes the arriving tuples into different classes. For each tuple class, a best plan is chosen. The choice of execution plans and the classifier are mutually dependent, so they cannot be decided one after another. They should be considered as a whole, meaning that the execution cost of each plan for each possible data subset should be estimated and compared. The search space is too big to use an enumerative search strategy, so the authors chose randomized search strategies. Similar to QM, correlation-aware multi-route stream query optimizer (CMR) [14] also partitions the data and computes an optimal plan for each partition. The difference is that it explores explicitly data correlations, which not only makes the partitioning more effective but also reduces the optimization complexity. HPE (Horizontal Partitioning with Eddies) [59] is another work using different plans for different data partitions. The main originalities are: first, the authors introduced the notion of conditional join plans (CJP), a new representation of search space which captures both the partitioning and the join orders for each partition combination; second, they use the eddy mechanism as the execution model, in order to share intermediate results between different plans.

4.3.3 Comparison

We compare the above methods in Table 6 using the five criteria. C1: They are all resistant to (or even take advantage of) non-uniform data distribution and data correlations. HPE uses eddies, so it is also resistant to data arrival delay and rate changing, etc. C2: They all optimize the current query. C3: They all focus on the classical optimization decisions: access methods, join methods and join order. Again, with eddies, HPE can also optimize the operator execution order. C4: The degradation risk is low, because characteristics of each sub-dataset are well-known by the optimizer. C5: The engineering cost is high, because both the optimizer and the plan executor need to be rewritten.

Table 6. Comparison of data partitioning methods

Criterion / Method	C1: Estim. error sources	C2:Target query types	C3: Target opt. decisions	C4:Deg. risk	C5:Eng. cost
ADP	DD,DC	C	AM,JM,JO	L	H
QM	DD,DC	C	AM,JM,JO	L	H
CMR	DD,DC	C	AM,JM,JO	L	H
HPE	All	C	AM,JM,JO,OEO	L	H

5. GLOBAL COMPARISON

In this section, we make a global comparison of the two approaches and their sub-classes.

With single-plan based approach, methods are easier to implement, but none of them can handle all types of estimation error sources; different methods could be combined to enlarge the application scope, but when there are too many uncertain factors, the degradation risk becomes high.

With multi-plan based approach, the degradation risk is limited, but the engineering cost is higher. Eddy-based methods can handle all kinds of estimation error sources, however, how to use eddies in highly parallel environment has not been well studied.

In table 7, we list briefly the advantages and limitations of the sub-classes for each approach.

Table 7. Global comparison

Approach	Class	Advantage	Limitation
Single-Plan Based	CI	Good for repeatedly-running queries	For current query, only JM, JO are optimized
	PM	Could be extended to improve all kinds of opt. decisions	May have high degradation risk
	RPS	Degradation risk is low if a robust plan exists	Difficult to handle too many uncertain factors
Multi-Plan Based	DPC	Easier to implement than TR and DP	AMC may consume too many resources
	TR	Deal with all kinds of estimation error sources	Parallelization problem is not addressed
	DP	Take advantage of inherent data characteristics	Optimization time may be long

6. CONCLUSION

Robust query optimization methods take into account the uncertainty of estimated parameter values, in order to avoid or recover from bad decisions caused by estimation errors.

In this paper, we classified the representative methods into two main approaches: single-plan based approach and multi-plan based approach. For each approach, we further differentiate the methods by categorizing them into subclasses. We analyzed and compared the methods of each subclass using five well-chosen criteria: estimation error sources, target query types, target optimization decisions, performance degradation risk and engineering cost. Finally, we gave a global comparison of the approaches and their subclasses.

The main conclusions that we have drawn are as follows: (1) different classes of methods in single-plan based approach can be combined to enlarge the application scope, (2) single-plan based approach is easier to be integrated into the main commercial DBMSs, but they only work well when there are very few uncertain parameters, and (3) accordingly, when there are too many uncertain parameters, the multi-plan based approach is a safer choice.

7. REFERENCES

[1] Aboulnaga, A. and Chaudhuri, S. 1999. Self-tuning Histograms: Building Histograms without Looking at Data. In *SIGMOD*. New York, USA, 181-192.

[2] Amsaleg, L., Franklin, M.J., Tomasic, A., and Urhan, T.1996. Scrambling Query Plans to Cope with Unexpected Delays. In *PDIS*. Miami, USA, 208-219.

[3] Antonshenkov, G. 1993. Dynamic Query Optimization in Rdb/VMS. In *ICDE*. Vienna, Austria, 538-547.

[4] Arcangeli, J.P., Hameurlain, A., Migeon, F., and Morvan, F. 2004. Mobile Agent Based Self-Adaptive Join for Wide-Area Distributed Query Processing. *Journal of Database Management*, 15(4): 25-44.

[5] Avnur, R. and Hellerstein, J.M. 2000. Eddies: Continuously Adaptive Query Processing. *SIGMOD*. Dallas, USA, 261-272.

[6] Babu, S. and Bizarro, P. 2005. Adaptive Query Processing in the Looking Glass. In *CIDR*. Asilomar, USA, 238-249.

[7] Babu, S., Bizarro, P., and Dewitt, D. 2005. Proactive Re-Optimization. In *SIGMOD*. Baltimore, USA, 107-118.

[8] Babcock, B. and Chaudhuri, S. 2005. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *SIGMOD*. Baltimore, USA, 119-130.

[9] Bizarro, P., Babu, S., DeWitt, D. and Widom, J. 2005. Content-Based Routing: Different Plans for Different Data. In *VLDB*. Trondheim, Norway, 757-768.

[10] Bizarro, P., Bruno, N., and DeWitt, D.J. 2009. Progressive Parametric Query Optimization. *KDE*, 21(4): 582 – 594.

[11] Bonneau, S. and Hameurlain, A. 1999. Hybrid Simultaneous Scheduling and Mapping in SQL Multi-Query Parallelization. In *DEXA*. Florence, Italy, 88-98.

[12] Bouganim, L., Fabret, F., Mohan, C., and Valduriez, P. 2000. Dynamic Query Scheduling in Data Integration Systems. In *ICDE*. San Diego, USA, 425-434.

[13] Bruno, N., Jain, S., and Zhou, J. 2013. Continuous Cloud-Scale Query Optimization and Processing. *PVLDB*, 6(11): 961-972.

[14] Cao, L. and Rundensteiner, E.A. 2013. High Performance Stream Query Processing With Correlation-Aware Partitioning. PVLDB, 7(4): 265-276.

[15] Chaudhuri, S., Narasayya, V., and Ramamurthy, R. 2008. A Pay-As-You-Go Framework for Query Execution Feedback. *PVLDB*, 1(1): 1141-1152.

[16] Chaudhuri, S. 2009. Query Optimizers: Time to Rethink the Contract? In *SIGMOD*. Providence, USA, 961-968.

[17] Chaudhuri, S., Narasayya, V., and Ramamurthy, R. 2009. Exact Cardinality Query Optimization for Optimizer Testing. *PVLDB*, 2(1): 994-1005.

[18] Chen, C.M. and Roussopoulos, N. 1994. Adaptive Selectivity Estimation Using Query Feedback. In *SIGMOD*. Minneapolis, USA, 161-172.

[19] Cole, R.L. and Graefe, G. 1994. Optimization of Dynamic Query Evaluation Plans. In *SIGMOD*. Minneapolis, USA, 150-160.

[20] Deshpande, A. 2004. An Initial Study of Overheads of Eddies. *SIGMOD Record*, 33(1): 44-49.

[21] Deshpande, A., Ives, Z.G., and Raman, V. 2007. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1): 1-140.

[22] Ergenç, B., Morvan, F., and Hameurlain, A. 2007. Robust Placement of Mobile Relational Operators for Large Scale Distributed Query Optimization. In *PDCAT*. Adelaide, Australia, 227-235.

[23] Evrendilek, C., Dogac, A., Nural, S., and Ozcan, F. 1997. Multidatabase Query Optimization. *Distributed and Parallel Databases*, 5(1):77-114.

[24] Ghazal, A., Seid, D., Crolotte, A., and Al-Kateb, M. 2012. Adaptive Optimizations of Recursive Queries in Teradata. In *SIGMOD*. Scottsdale, USA, 851-860.

[25] Graefe, G. and Ward, K. 1989. Dynamic query evaluation plans. In *SIGMOD*. Portland, USA, 358-366.

[26] Graefe, G., Kuno, H.A., and Wiener, J.L. 2009. Visualizing the Robustness of Query Execution. In *CIDR*. Asilomar, USA.

[27] Graefe, G., König, A.C., Kuno, H., Markl, V., and Sattler, K.U. 2010. Robust Query Processing. *Dagstuhl Workshop Summary 10381*, Wadern, Germany.

[28] Graefe, G. 2011. Robust Query Processing (Research Panel). In *ICDE*. Hannover, Germany, 1361.

[29] Graefe, G., Guy, W., Kuno, H.A., and Paulley G. 2012. Robust Query Processing. *Dagstuhl Workshop Summary 12321*, Wadern, Germany.

[30] Gounaris, A., Paton, N.W., Fernandes, A., and Sakellariou, R. 2002. Adaptive Query Processing: A Survey. In *BNCOD*. Sheffield, UK, 11-25.

[31] Gounaris, A., Tsamoura. E., and Manolopoulos, Y. 2013. Adaptive Query Processing in Distributed Settings. *Advanced Query Processing*, Vol. 1: 211-236.

[32] Hameurlain, A. and Morvan, F. 2002. CPU and Incremental Memory Allocation in Dynamic Parallelization of SQL Queries. *Journal of Parallel Computing*, 28(4): 525-556.

[33] Han, W., Ng, J., Markl, V., Kache, H, and Kandil, M. 2007. Progressive Optimization in a Shared-Nothing Parallel Database. In *SIGMOD*. Beijing, China, 809-820.

[34] Harish, D., Darera, P., and Harista, J. 2007. On the Production of Anorexic Plan Diagrams. In VLDB. Vienna, Austria, 1081-1092.

[35] Harish, D., Darera, P., and Harista, J. 2008. Identifying Robust Plans through Plan Diagram Reduction. *PVLDB*, 1(1): 1124-1140.

[36] Herodotou, H. and Babu, S. Xplus. 2010. A SQL-Tuning-Aware Query Optimizer. *PVLDB*, 3(1): 1149-1160.

[37] Hong, W. and Stonebraker, M.1993. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1): 9–32.

[38] Ioannidis, Y. and Christodoulakis, S. 1991. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*. Denver, USA, 168-177.

[39] Ioannidis, Y., Ng, R.T., Shim, K., and Sellis, T.K. 1997. Parametric query optimization. *VLDB Journal*, 6(2): 132-151.

[40] Ioannidis, Y. 2003. The History of Histograms (abridged). In *VLDB*. Berlin, Germany, 19-30.

[41] Ives, Z. G., Florescu, D., Friedman, M., Levy, A.Y., and Weld, D.S. 1999. An Adaptive Query Execution System for Data Integration. In *SIGMOD*. Philadelphia, USA, 299-310.

[42] Ives, Z.G., Halevy, A.Y., and Weld, D.S. 2004. Adapting to Source Properties in Processing Data Integration Queries. In *SIGMOD*. Paris, France, 395-406.

[43] Kabra, N. and DeWitt, D. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD*. Seattle, USA, 106-117.

[44] Larson, P., Lehner, W., Zhou, J., and Zabback, P. 2007. Cardinality Estimation Using Sample Views with Quality Assurance. In *SIGMOD*. Beijing, China, 175-186.

[45] Lipton, R.J., Naughton, J.F., and Schneider, D.A. 1990. Practical Selectivity Estimation through Adaptive Sampling. In *SIGMOD*. Atlantic City, USA, 1-11.

[46] Markl, V., Raman, V., Simmen, D. E., Lohman, G.M., and Pirahesh, H. 2004. Robust Query Processing through Progressive Optimization. *SIGMOD*. Paris, France, 659-670.

[47] Markl, V., Haas, P.J., Kutsch, M., Megiddo, N., Srivastava, U., and Tran, T.M. 2007. Consistent Selectivity Estimation via Maximum Entropy. *VLDB Journal*, 16(1): 55-76.

[48] Morvan, F. and Hameurlain, A. 2009. Dynamic Query Optimization: Towards Decentralized Methods. *International Journal of Intelligent Information and Database Systems*, 4(3): 461-482.

[49] Neumann, T. and Calindo-Legaria, C. 2013.Taking the Edge off Cardinality Estimation Errors using Incremental Execution. In *BTW*. Magdeburg, Germany, 73-92.

[50] Nehme, R.V., Works, K.E., Rundensteiner, E.A., and Bertino, E. 2009. Query Mesh: Multi-Route Query Processing Technology (Demo). *PVLDB*, 2(2): 1530-1533.

[51] Nehme, R.V., Works, K., Lei, C., Rundensteiner, E.A., and Bertino, E. 2013. Multi-Route Query Processing and Optimization. Journal of Computer and System Sciences, 79(3): 312-329.

[52] Olken, F. and Rotem, D. 1986. Simple Random Sampling from Relational Databases. In *VLDB*. Kyoto, Japan, 160-169.

[53] Picasso Database Query Optimizer Visualizer. http://dsl.serc.iisc.ernet.in/projects/PICASSO/

[54] Raman, V., Deshpande, A., and Hellerstein, J.M. 2003. Using State Modules for Adaptive Query Processing. In *ICDE*. Bangalore, India, 353-364.

[55] Reddy, N. and Harista, J. 2005. Analyzing Plan Diagrams of Database Query Optimizers. In *VLDB*. Trondheim, Norway, 1228-1239.

[56] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., and Price, T.G. 1979. Access Path Selection in a Relational DBMS. In *SIGMOD*. Boston, USA, 23-34.

[57] Stillger, M., Lohman, G., Markl, V., and Kandil, M.2001. LEO-DB2's Learning Optimizer. *VLDB*. Roma, Italy, 19-28.

[58] Tian, F. and DeWitt, D.J. 2003. Tuple Routing Strategies for Distributed Eddies. In VLDB. Berlin, Germany, 333-344.

[59] Tzoumas, K., Deshpande, A., and Jensen, C.S. 2010. Sharing-Aware Horizontal Partitioning for Exploiting Correlations during Query Processing. *PVLDB*, 3(1): 542-553.

[60] Tzoumas, K., Deshpande, A., and Jensen, C.S. 2011. Lightweight Graphical Models for Selectivity Estimation without Independence Assumptions. *PVLDB*, 4(11): 852-863.

[61] Tzoumas, K., Deshpande, A., and Jensen, C.S. 2013. Efficient Adapting Graphical Models for Selectivity Estimation. *VLDB Journal*, 22(1): 3-27.

[62] Wilschut, A. N. and Apers, P. M. G. 1991. Dataflow Query Execution in a Parallel Main-Memory Environment. In PDIS, Miami Beach, USA, 68–77.

[63] Wiener, J.L., Kuno, H., and Graefe, G. 2009. Benchmarking Query Execution Robustness. In TPC Technology Conference on Performance Evaluation & Benchmarking. Lyon, France, 153-166.

[64] Zhou, Y., Ooi, B.C., Tan, K.L., and Tok, W.H. 2005. An Adaptable Distributed Query Processing Architecture. *Knowledge and Data Engineering*. 53(3): 283-309.