IJITKM 0973-4414 Volume-9, Number-1 Jun-Dec 2015 pp. 62-95 **Impact Factor 2.8** available online at www.csjournalss.com

A Survey on Query Optimization for Databases

Swapan Kumar Samaddar [1], Supreethi Pujari[2]
[1] Genpact, Hyderabad, India
[2] JNTU, Hyderabad, India
swapansamaddar@gmail.com, supreethi.pujari@gmail.com

Abstract: The automated optimization of declarative SQL queries is a classical problem that has been diligently addressed by the database community over several decades. However, due to its inherent complexities and challenges, the topic has largely remained a "black art", and the quality of the query optimizer continues to be a key differentiator between competing database products, with large technical teams involved in their design and implementation. Over the past few years, a fresh perspective on the behavior of modern query optimizers has arisen through the introduction and development of the "plan diagram" concept. A plan diagram is a visual representation of the plan choices made by the optimizer over a space of input parameters, such as relational selectivities. we provide a detailed walk-through of plan diagrams, their processing, and their applications. We begin by showcasing a variety of plan diagrams that provide intriguing insights into current query optimizer implementations. A suite of techniques for efficiently producing plan diagrams are then outlined. Subsequently, we present a suite of post-processing algorithms that take optimizer plan diagrams as input, and output new diagrams with demonstrably superior query processing characteristics, such as robustness to estimation errors. Following up, we explain how these offline characteristics can be internalized in the query optimizer, resulting in an intrinsically improved optimizer that directly produces high quality plan diagrams. Finally, we enumerate a variety of open technical problems, and promising future research directions. All the plan diagrams are sourced from popular industrial-strength query optimizers operating on benchmark decision-support environments, and will be graphically displayed on the Picasso visualization platform.

Keyword: DBridge, Picasso, TPC-H, TPC-DS, CostGreedy, ThresholdGreedy, SEER, LiteSEER.

Introduction

Relational query languages provide a high-level "declarative" interface to access data stored in relational databases. Over time, SQL [12] has emerged as the standard for relational query languages. Two key components of the query evaluation component of a SQL database system are the query optimizer and the query execution engine. The query execution engine implements a set of physical operators. An operator takes as input one or more data streams and produces an output data stream. Examples of physical operators are (external) sort, sequential scan, index scan, nested loop join, and sort-merge join. The simplest way to think of physical operators is as pieces of code that are used as building blocks to make possible the execution of SQL queries. An abstract representation of such an execution is a physical operator tree, as illustrated in Figure 1. The edges in an operator tree represent the data flow among the physical operators. The execution engine is responsible for the execution of the plan that results in generating answers to the query. Therefore, the capabilities of the query execution engine determine the structure of the operator trees that are feasible.

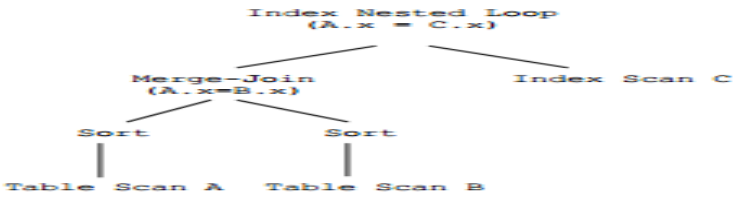


Figure 1. Operator Tree

DOI: 10.141079/IJITKM.2015.909

The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of a SQL query as input and is responsible for generating an efficient execution plan for the given SQL query from the space of possible execution plans. The task of an optimizer is nontrivial since for a given SQL query, there can be a large number of possible operator trees:

• The algebraic representation of the given query can be transformed into many other logically equivalent algebraic representations: e.g.,

Join(Join(A,B),C)= Join(Join(B,C),A)

• For a given algebraic representation, there may be many operator trees that implement the algebraic expression, e.g., typically there are several join algorithms supported in a database system. Furthermore, the throughput or the response times for the execution of these plans may be widely different. Therefore, a judicious choice of an execution by the optimizer is of critical importance. Thus, query optimization can be viewed as a difficult search problem. In order to solve this problem, we need to provide:

• A space of plans (search space).

• A cost estimation technique so that a cost may be assigned to each plan in the search space. Intuitively, this is an estimation of the resources needed for the execution of the plan.

• An enumeration algorithm that can search through the execution space. A desirable optimizer is one where (1) the search space includes plans that have low cost(2) the costing technique is accurate(3) the enumeration algorithm is efficient. Each of these three tasks is nontrivial and that is why building a good optimizer is an enormous undertaking. We begin by discussing the System-R optimization framework since this was a remarkably elegant approach that helped fuel much of the subsequent work in optimization.

AN EXAMPLE: SYSTEM-R OPTIMIZER

The System-R project significantly advanced the state of query optimization of relational systems. The ideas in [13] have been incorporated in many commercial optimizers continue to be remarkably relevant. I will present a subset of those important ideas here in the context of Select-Project-Join (SPJ) queries. The class of SPJ queries is closely related to and encapsulates conjunctive queries, which are widely studied in Database Theory. The search space for the System-R optimizer in the context of a SPJ query consists of operator trees that correspond to linear sequence of join operations, e.g., the sequence Join(Join(Join(A,B),C),D) is illustrated in Figure 2(a). Such sequences are logically equivalent because of associative and commutative properties of joins. A join operator can use either the nested loop or sort-merge implementation. Each scan node can use either index scan (using a clustered or non clustered index) or sequential scan. Finally, predicates are evaluated as early as possible. The cost model assigns an estimated cost to any partial or complete plan in the search space. It also determines the estimated size of the data stream for output of every operator in the plan. It relies on:

(a) A set of statistics maintained on relations and indexes, e.g., number of data pages in a relation, number of pages in an index, number of distinct values in a column

(b) Formulas to estimate selectivity of predicates and to project the size of the output data stream for every operator node. For example, the size of the output of a join is estimated by taking the product of the sizes of the two relations and then applying the joint selectivity of all applicable predicates.

(c) Formulas to estimate the CPU and I/O costs of query execution for every operator. These formulas take into account the statistical properties of its input data streams, existing access methods over the input data streams, and any available order on the data stream (e.g., if a data stream is ordered, then the cost of a sort-merge join on that stream may be significantly reduced). In addition, it is also checked if the output data stream will have any order. The cost model uses (a)-(c) to compute and associate the following information in a bottom-up fashion for operators in a plan: (1) The size of the data stream represented by the output of the operator node. (2) Any ordering of tuples created or sustained by the output data stream of the operator node. (3) Estimated execution cost for the operator (and the cumulative cost of the partial plan so far).

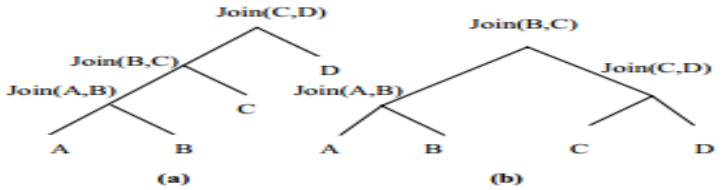


Figure 2. (a) Linear and (b) bushy join

The enumeration algorithm for System-R optimizer demonstrates two important techniques: use of dynamic programming and use of interesting orders. The essence of the dynamic programming approach is based on the assumption that the cost model satisfies the principle of optimality. Specifically, it assumes that in order to obtain an optimal plan for a SPJ query Q consisting of k joins, it suffices to consider only the optimal plans for sub expressions of Q that consist of (k-1) joins and extend those plans with an additional join. In other words, the suboptimal plans for sub expressions of Q (also called sub queries) consisting of (k-1) joins do not need to be considered further in determining the optimal plan for Q. Accordingly, the dynamic programming based enumeration views a SPJ query Q as a set of relations {R1,..Rn} to be joined. The enumeration algorithm proceeds bottom-up. At the end of the j-th step, the algorithm produces the optimal plans for all sub queries of size j. To obtain an optimal plan for a sub query consisting of (j+1) relations, we consider all possible ways of constructing a plan for the sub query by extending the plans constructed in the jth step. For example, the optimal plan for {R1,R2,R3,R4} is obtained by picking the plan with the cheapest cost from among the optimal plans for: (1) Join({R1,R2,R3},R4) (2) Join({R1,R2,R4},R3) (3) Join ({R1,R3,R4},R2) (4) Join({R2,R3,R4}, R1). The rest of the plans for {R1,R2,R3,R4} may be discarded. The dynamic programming approach is significantly faster than the naïve approach since instead of O(n!) plans, only O(n2 n -1) plans need to be enumerated. The second important aspect of System R optimizer is the consideration of interesting orders. Let us now consider a query that represents the join among {R1,R2,R3} with the predicates R1.a = R2.a = R3.a. Let us also assume that the cost of the plans for the sub query {R1,R2} are x and y for nested-loop and sort-merge join respectively and x <y. In such a case, while considering the plan for {R1, R2, R3}, we will not consider the plan where R1 and R2 are joined using sort-merge. However, note that if sort-merge is used to join R1 and R2, the result of the join is sorted on a. The sorted order may significantly reduce the cost of the join with R3. Thus, pruning the plan that represents the sort merge join between R1and R2 can result in sub-optimality of the global plan. The problem arises because the result of the sort merge join between R1 and R2 has an ordering of tuples in the output stream that is useful in the subsequent join. However, the nested-loop join does not have such ordering. Therefore, given a query, System R identified ordering of tuples that are potentially consequential to execution plans for the query (hence the name interesting orders).

SEARCH SPACE

The search space for optimization depends on the set of algebraic transformations that preserve equivalence and the set of physical operators supported in an optimizer. The optimizer may use several representations of a query during the lifecycle of optimizing a query. The initial representation is often the parse tree of the query and the final representation is an operator tree. An intermediate representation that is also used is that of logical operator trees (also called query trees) that captures an algebraic expression. Figure 2 is an example of a query tree. Often, nodes of the query trees are annotated with additional information. For SPJ queries, such a structure is often captured by a query graph where nodes represent relations (correlation variables) and labeled edges represent join predicates among the relations (see Figure 3). Although conceptually simple, such a representation falls short of representing the structure of arbitrary SQL statements in a number of ways.

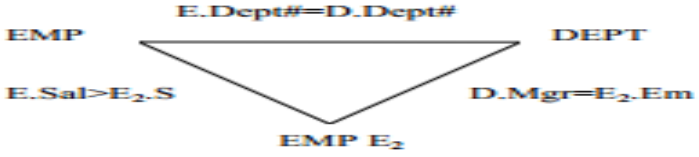


Figure 3. Query

Generalizing Join Sequencing

In many of the systems, the sequence of join operations is syntactically restricted to limit search space. For example, in the System R project, only linear sequences of join operations are considered and Cartesian product among relations is deferred until after all the joins. Since join operations are commutative and associative, the sequence of joins in an operator tree need not be linear. In particular, the query consisting of join among relations R1,R2,R3,R4 can be algebraically represented and evaluated as Join(Join(A,B),Join(C,D)). Such query trees are called bushy, illustrated in Figure 2(b). Bushy join sequences require materialization of intermediate relations. While bushy trees may result in cheaper query plan, they expand the cost of enumerating the search space considerably Although there has been some studies of merits of exploring the bushy join sequences, by and large most systems still focus on linear join sequences and only restricted subsets of bushy join trees. Deferring Cartesian products may also result in poor performance. In an extensible system, the behavior of the join enumerator may be adapted on a per query basis so as to restrict the "bushy"-ness of the join trees and to allow or disallow Cartesian products [14]. However, it is nontrivial to determine a priori the effects of such tuning on the quality and cost of the search.

Outer join and Join

One-sided outer join is an asymmetric operator in SQL that preserves all of the tuples of one relation. Symmetric outer joins preserve both the operand relations. Thus, (R LOJ S), where LOJ designates left outer join between R and S, preserves all tuples of R. In addition to the tuples from natural join, the above operation contains all remaining tuples in R that fail to join with S (padded with NULLs for their S attributes). Unlike natural joins, a sequence of outer joins and joins do not freely commute. However, when the join predicate is between (R,S) and the outer join predicate is between (S,T), the following identity holds: Join(R, S LOJ T) = Join (R,S) LOJ .T If the above associative rule can be repeatedly applied, we obtain an equivalent expression where evaluation of the "block of joins" precedes the "block of outer joins". Subsequently, the joins may be freely reordered among themselves

Group-By and Join

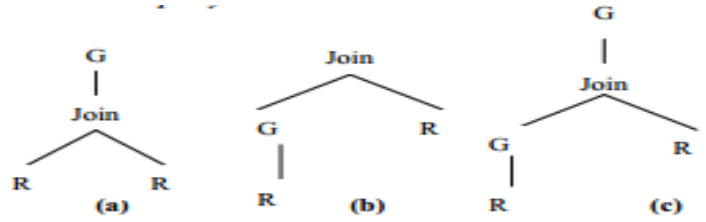


Figure 4. Group By and Join

In traditional execution of a SPJ query with group-by, the evaluation of the SPJ component of the query precedes the group by. The set of transformations described in this section enable the group by operation to precede a join. These transformations are applicable to queries with SELECT DISTINCT since the latter is a special case of group-by. Evaluation of a group-by operator can potentially result in a significant reduction in the number of tuples, since only one tuple is generated for every partition of the relation induced by the group-by operator. Therefore, in some cases, by first doing the group-by, the cost of the join may be significantly reduced. Moreover, in the presence of an appropriate index, a group-by operation may be evaluated inexpensively. A dual of such transformations corresponds to the case where a group-by operator may be pulled up past a join. Consider the query tree in Figure 4(a). Let the join between R1and R2 be a foreign key join and let the aggregated columns of Gbe from columns in R1 and the set of group-by columns be a superset of the foreign key columns of R1. For such a query, let us consider the corresponding operator tree in Fig. 4(b), where G1=G. In that tree, the final join with R2 can only eliminate a set of potential partitions of R1 created by G1 but will not affect the partitions nor the aggregates computed for the partitions by G1 since every tuple in R1 will join with at mostone tuple in R2. Therefore, we can push down the group-by, as shown in Fig. 4(b) and preserve equivalence for arbitrary side-effect free aggregate functions. Fig. 4(c) illustrates an example where the transformation introduces a group-by and represents a class of useful examples where the group-by operation is done in stages.

Reducing Multi-Block Queries to Single Block

The technique described in this section shows how under some conditions, it is possible to collapse a multi-block SQL query into a single block SQL query.

Merging Views

Let us consider a conjunctive query using SELECT ANY. If one or more relations in the query are views, but each is defined through a conjunctive query, then the view definitions can simply be "unfolded" to obtain a single block SQL query. For example, if a query Q = Join(R,V)and view V = Join(S,T), then the query Q can be unfolded to Join(R, Join(S,T))and may be freely reordered. Such a step may require some renaming of the variables in the view definitions. Unfortunately, this simple unfolding fails to work when the views are more complex than simple SPJ queries. When one or more of the views contain SELECT DISTINCT, transformations to move or pull up DISTINCT need to be careful to preserve the number of duplicates correctly [15].

Merging Nested Sub queries

Consider the following example of a nested query from [16] where Emp# and Dept# are keys of the corresponding relations:

SELECT Emp.Name FROM Emp WHERE Emp.Dept# IN
SELECT Dept.Dept# FROM Dept WHERE Dept.Loc='Denver' AND Emp.Emp# = Dept.Mgr

If tuple iteration semantics are used to answer the query, then the inner query is evaluated for each tuple of the Dept relation once. An obvious optimization applies when the inner query block contains no variables from the outer query block (uncorrelated). In such cases, the inner query block needs to be evaluated only once. However, when there is indeed a variable from the outer block, we say that the query blocks are correlated. For example, in the query above, Emp. Emp# acts as the correlated variable. The above nested query reduces to:

SELECT E.Name FROM Emp E, Dept D WHERE E.Dept# = D.Dept# AND D.Loc = 'Denver'AND E.Emp# = D.Mgr

The complexity of the problem depends on the structure of the nesting, i.e., whether the nested sub query has quantifiers (e.g., ALL, EXISTS), aggregates or neither. In the simplest case, of which the above query is an example observed that the tuple semantics can be modeled as Semi join(Emp,Dept, Emp.Dept# = Dept.Dept#)

Once viewed this way, it is not hard to see why the query may be merged since:

Semi join(Emp,Dept,Emp.Dept# = Dept. Dept#) = Project(Join(Emp,Dept), Emp.*) Where Join(Emp,Dept)is on the predicate Emp.Dept# = Dept. Dept#. The second argument of the Project operator indicates that all columns of the relation Emp must be retained. The problem is more complex when aggregates are present in the nested sub query, since merging query blocks now requires pulling up the aggregation without violating the semantics of the nested query:

SELECT Dept.name FROM Dept WHERE Dept.num-of-machines ≥ (SELECT COUNT(Emp.*) FROM Emp WHERE Dept.name= Emp.Dept_name)

It is especially tricky to preserve duplicates and nulls. To appreciate the subtlety, observe that if for a specific value of Dept.name(say d), there are no tuples with a matching Emp.Dept_name, i.e., even if the predicate Dept.name= Emp.dept_namefails, then there is still an output tuple for the Dept tuple d. However, if we were to adopt the transformation used in the first query of this section, then there will be no output tuple for the dept d since the join predicate fails. Therefore, in the presence of aggregation, we must preserve all the tuples of the outer query block by a left outer join. In particular, the above query can be correctly transformed to:

SELECT Dept.name FROM Dept LEFT OUTER JOIN Emp ON (Dept.name= Emp.dept_name) GROUP BY Dept.name HAVING Dept. num-of-machines < COUNT (Emp.*)

Using Semi join Like Techniques for Optimizing Multi-Block Queries :

The goal of the approach described in this section is to exploit the selectivity of predicates across blocks. It is conceptually similar to the idea of using semi join to propagate from a site A to a remote site B information on relevant values of A so that B sends to A no unnecessary tuples. In the context of multi-block queries, A and B are in different query blocks but are parts of the same query and therefore the transmission cost is not an issue.

CREATE VIEW DepAvgSal As (SELECT E.did, Avg(E.Sal) AS avgsal FROM Emp E GROUP BY E.did) SELECT E.eid, E.sal FROM Emp E, Dept D, DepAvgSal V WHERE E.did = D.did AND E.did = V.did AND E.age < 30 AND D.budget > 100k AND E.sal > V.avgsal

The technique recognizes that we can create the set of relevant E.did by doing only the join between E and Din the above query and projecting the unique E.did. This set can be passed to the view DepAvgSalto restrict its computation. This is accomplished by the following three views.

CREATE VIEW partialresult AS (SELECT E.id, E.sal, E.did FROM Emp E, Dept D WHERE E.did=D.did AND E.age < 30 AND D.budget > 100k)

CREATE VIEW Filter AS (SELECT DISTINCT P.did FROM PartialResult P)

CREATE VIEW LimitedAvgSal AS (SELECT E.did, Avg(E.Sal) AS avgsal FROM Emp E, Filter F WHERE E.did = F.did GROUP BY E.did)

The reformulated query on the next page exploits the above views to restrict computation.

SELECT P.eid, P.sal FROM PartialResult P, LimitedDepAvgSal V WHERE P.did = V.did AND P.sal > V.avgsal

The above technique can be used in a multi-block query containing view (including recursive view) definitions or nested sub queries.

ENUMERATION ARCHITECTURES

An enumeration algorithm must pick an inexpensive execution plan for a given query by exploring the search space. The System R join enumerator was designed to choose only an optimal linear join order. A software engineering consideration is to build the enumerator so that it can gracefully adapt to changes in the search space due to the addition of new transformations, the addition of new physical operators (e.g., a new join implementation) and changes in the cost estimation techniques. More recent optimization architectures have been built with this paradigm and are called extensible optimizers. Building an extensible optimizer is a tall order since it is more than simply coming up with a better enumeration algorithm. Rather, they provide an infrastructure for evolution of optimizer design. However, generality in the architecture must be balanced with the need for efficiency in enumeration. We focus on two representative examples of such extensible optimizers: Starburst, Volcano/Cascades, DBridge and Picasso briefly.

Starburst

Query optimization in the Starburst project [17] begins with a structural representation of the SQL query that is used throughout the lifecycle of optimization. This representation is called the Query Graph Model (QGM). In the QGM, a box represents a query block and labeled arcs between boxes represent table references across blocks. Each box contains information on the predicate structure as well as on whether the data stream is ordered. In the query rewrite phase of optimization rules are used to transform a QGM into another equivalent QGM. Rules are modeled as pairs of arbitrary functions. The first one checks the condition for applicability and the second one enforces the transformation. A forward chaining rule engine governs the rules. Rules may be grouped in rule classes and it is possible to tune the order of evaluation of rule classes to focus search. Since any application of a rule results in a valid QGM, any set of rule applications guarantee query equivalence (assuming rules themselves are valid). The query rewrite phase does not have the cost information available. This forces this module to either retain alternatives obtained through rule application or to use the rules in a heuristic way (and thus compromise optimality). The second phase of query optimization is called plan optimization. In this phase, given a QGM, an execution plan (operator tree) is chosen. In Starburst, the physical operators (called LOLEPOPs) may be combined in a variety of ways

to implement higher level operators. In Starburst, such combinations are expressed in a grammar production-like language . The realization of a higher-level operation is expressed by its derivation in terms of the physical operators. In computing such derivations, comparable plans that represent the same physical and logical properties but have higher costs, are pruned. Each plan has a relational description that corresponds to the algebraic expression it represents, an estimated cost, and physical properties (e.g., order). These properties are propagated as plans are built bottom-up. Thus, with each physical operator, a function is associated that shows the effect of the physical operator on each of the above properties. The join enumerator in this system is similar to System-R's bottom-up enumeration scheme.

Volcano/Cascades

In these systems, rules are used universally to represent the knowledge of search space. Two kinds of rules are used. The transformation rules map an algebraic expression into another. The implementation rules map an algebraic expression into an operator tree. The rules may have conditions for applicability. Logical properties, physical properties and costs are associated with plans. The physical properties and the cost depend on the algorithms used to implement operators and its input data streams. For efficiency, Volcano/Cascades uses dynamic programming in a top-down way ("memoization"). When presented with an optimization task, it checks whether the task has already been accomplished by looking up its logical and physical properties in the table of plans that have been optimized in the past. Otherwise, it will apply a logical transformation rule, an implementation rule, or use an enforcer to modify properties of the data stream. At every stage, it uses the promise of an action to determine the next move. The promise parameter is programmable and reflects cost parameters. The Volcano/Cascades framework differs from Starburst in its approach to enumeration: (a) These systems do not use two distinct optimization phases because all transformations are algebraic and cost-based. (b) The mapping from algebraic to physical operators occurs in a single step. (c) Instead of applying rules in a forward chaining fashion, as in the Starburst query rewrite phase, Volcano/Cascades does goal-driven application of rules.

DBridge

DBridge is a program transformation tool. DBridge works on Java applications that use JDBC API [18] to access database. The tool performs static analysis of the input program and identifies opportunities for replacing iterative database access with set oriented access; it then rewrites the application code and the embedded queries together for set oriented processing. DBridge is designed to be a source-to-source transformation tool, and to this end, it ensures readability and maintainability of the transformed code. The tool is thus best suited for integration into an application development environment (IDE). DBridge can also be used as a preprocessing step inside a language compiler, thus making the compiler "database access aware".

OVERVIEW OF DBRIDGE

As an illustration of the kind of transformations DBridge can perform, consider the Java program snippet shown in Figure 5.

```
Connection con = DriverManager.getConnection(url);
PreparedStatement pstmt = con.prepare(
    "SELECT count(partkey) FROM part WHERE category=?");
while(category != -1) {
    pstmt.setInt(1, category);
    ResultSet rs = pstmt.executeQuery();
    if (rs.next()) {
        partCount = rs.getInt(0);
        total += partCount;
    }
    category = getParent(category);
}
```

Figure 5: A Program Snippet with JDBC Calls

```
Connection con = DriverManager.getConnection(dbrUrl);
PreparedStatement pstmt = con.prepare(
  "SELECT count(partkey) FROM part WHERE category=?");
while(category != -1) {
    pstmt.setInt(1, category);
    pstmt.addBatch();
    category = getParent(category);
}
pstmt.executeBatch();
while (pstmt.getMoreResults()) {
    ResultSet rs = pstmt.getResultSet();
    if (rs.next()) {
        partCount = rs.getInt(0);
        total += partCount;
    }
}
```

Figure 6: JDBC Example after Transformation

The program computes the total number of parts in a given category and all its parent categories. Note the repeated execution of a parameterized aggregate query inside the while loop. The program snippet after transformation by DBridge is shown in Figure 6. The transformed code is a result of the application of several transformation rules. We give an overview of some of the important transformations with the help of the example.

Statement Reordering

DBridge applies a set of transformation rules to reorder the statements within the loop body, so as to permit set oriented execution. In Figure 6, note that the statement invoking getParent, is moved up ahead of the query execution. Statement reordering is performed taking inter-statement data dependencies into account. DBridge can also introduce temporary variables to break certain inter-statement data dependencies that otherwise prohibit set oriented execution;

Loop Splitting

This is the key transformation to enable set oriented execution. In Figure 6, the loop in the original program is split into two parts. The first loop in the transformed program generates all the parameter bindings. Next, a rewritten form of the query is executed to obtain results for all the parameter bindings together. Then, the second loop executes statements that depend on the query results.

Query Rewrite

After collecting all the parameter bindings, the program calls the executeBatch method, which internally transforms the query statement into a set oriented form, which is often more efficient. For example, the scalar aggregate query in the example would be transformed into the following query , where pb is a temporary table in which the parameter bindings are materialized.

SELECT pb.category, le.c1 FROM pbatch pb, OUTER APPLY (SELECT count(partkey) as c1 FROM part WHERE category=pb.category) le;

The rewritten query uses the OUTER APPLY construct of Microsoft SQL Server but can instead be written using a left outer join combined with the LATERAL construct of SQL:99. Most widely used database systems can unnest such a query into a form that uses joins or outer joins. For example, the unnested form of the above query could be:

SELECT pb.category, count(partkey) FROM pbatch pb LEFT OUTER JOIN part p ON pb.category=p.category GROUP BY pb.category;

Such a rewriting enables the use of efficient set oriented query processing algorithms such as hash or merge join.

Rewrite of Conditional Blocks

DBridge can deal with conditional control transfer statements (if-then-else), and query execution statements inside conditional blocks. DBridge also handles order-sensitive operations within the loop correctly. Order-sensitive operations are operations whose order of execution is important for the correctness of the program. We illustrate these two features using Figure 7. The program, after transformation, is shown in Figure 8.

```
while(category != -1) {
    if(isActive(category)) {
        pstmt.bind(1, category);
        rs = pstmt.executeQuery();
        rs.next();
        partCount = rs.getInt(0);
        total += partCount;
        print(category, partCount);
    }
    category = getParent(category);
}
```

Figure 7: Queries inside Conditional Blocks

DBridge first transforms conditional blocks into a sequence of guarded statements by introducing a boolean variable to remember the branching decision. It then applies the loop splitting transformation and replaces repeated execution of the query with a single invocation of its set oriented form. Finally, the sequence of guarded statements are merged back to have conditional blocks, as can be seen in the second loop of Figure 8.

```
LoopContextTable ctx;
while(category != -1) {
    boolean f = isActive(category);
    if(f) {
        pstmt.bind(1, category);
        pstmt.addBatch();
    }
    tempCat = category;
    category = getParent(category);
    ctx.addRecord(new Record(loopKey, f, tempCat));
}
pstmt.executeBatch();
// Now, read all the results and augment ctx with a new
// column partCount to hold the part count for each category.
ctx.mergeResults(pstmt);
for (Record r: ctx) { // We assume ctx to be an ordered table
    if(r.f) {
        total += r.partCount;
        print(r.tempCat, r.partCount);
    }
}
```

Figure 8: Transformation of Figure 7

Nested Loops

A query execution statement can be inside a loop, which is nested within another loop. DBridge works with arbitrary levels of loop nesting.

SYSTEM DESIGN AND IMPLEMENTATION

DBridge is designed to meet the following requirements:

• Semantics Preservation: For a program transformation tool like DBridge, ensuring correctness is a strict requirement. DBridge ensures that the transformed program is equivalent in its functionality to the original program. DBridge's transformations are based on a set of formally defined equivalence rules.

• Extensibility: DBridge is designed in a way that provides an elegant framework for introducing new transformation rules or extending existing rules. Each rule is encapsulated as an object, and all the information necessary to apply a

rule is provided by the framework via the program dependence graph. The important phases in the program transformation process are shown in Figure 9. The input Java source file is first converted into an intermediate representation, on which we perform data flow analysis. Using this analysis, we construct a Dependence Graph, which is the basic data structure on which our transformation rules rely. The main task of our program transformation tool appears in the Apply Trans Rules

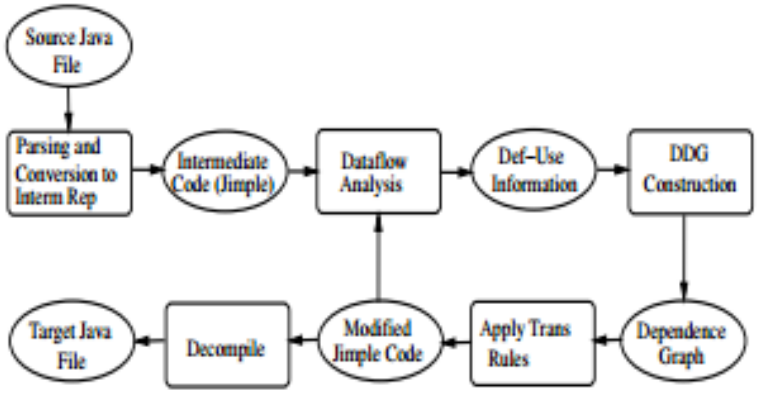


Figure 9. Program Transformation Phases

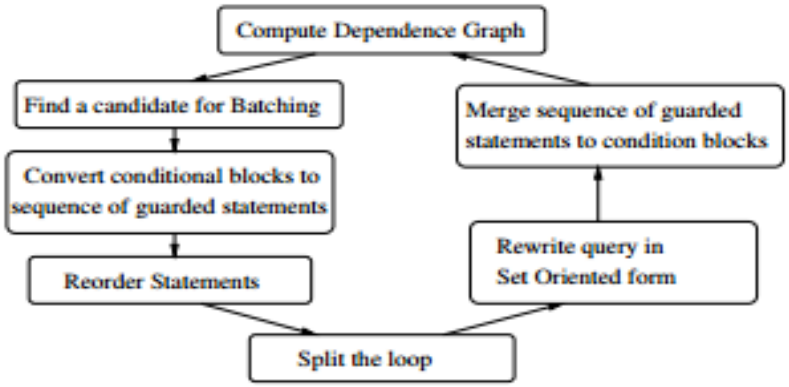


Figure 10. Iterative application of Transformation rules

phase. The program transformation rules are applied in an iterative manner, updating the dataflow information each time the code changes. The rule application process stops when all (or the user chosen) query execution statements within loops are transformed into their corresponding set oriented forms. Once all the transformations are done, the intermediate representation is converted back to a target Java source file.

PICASSO

The role of query optimizers has become especially critical during this decade due to the high degree of processing complexity characterizing current data warehousing and mining applications, as exemplified by the TPC-H and TPC-DS [1,2] decision support benchmarks. There is a visualization tool, called Picasso [3], for graphically profiling and analyzing the behavior of database query optimizers. The tool is operational on a rich set of industrial-strength optimizers, including IBM DB2, Microsoft SQL Server, Oracle, Sybase ASE and PostgreSQL. It has been employed as

• a query optimizer analysis, debugging, and redesign aid by system developers;

• a query optimization test-bed by database researchers; and

• a query optimizer pedagogical support by database instructors and students.

PICASSO DIAGRAMS

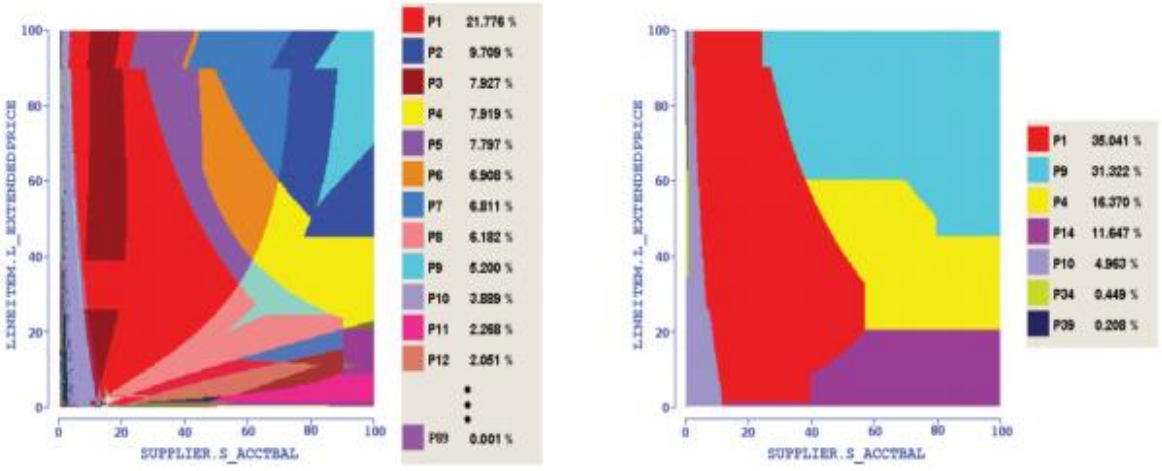
Given a parametrized SQL query template that defines a relational selectivity space, and a choice of database engine, the Picasso tool automatically generates a variety of diagrams that characterize the behavior of the engine's optimizer over this space. For example, the so-called "Plan Diagram" [4], representing a color coded pictorial enumeration of the plan choices made by the optimizer over the selectivity space. Specifically, plan diagrams visually capture the optimality regions of POSP [19], the parametric optimal set of plans. To make these notions concrete, consider QT8, the parametrized 2D query template shown in Figure 11, based on Query 8 of the TPCH benchmark. Here, selectivity variations on the SUPPLIER and LINEITEM relations are specified through the s acctbal :varies and l extendedprice :varies predicates, respectively.

```
select o_year, sum(case when nation = 'BRAZIL' then volume else 0
       end) / sum(volume) as mkt_share
from (select YEAR(o_orderdate) as o_year, l_extendedprice * (1 -
       l_discount) as volume, n2.n_name as nation
       from part, supplier, lineitem, orders, customer,
            nation n1, nation n2, region
       where p_partkey = l_partkey and s_suppkey = l_suppkey
            and l_orderkey = o_orderkey and o_custkey = c_custkey
            and c_nationkey = n1.n_nationkey and n1.n_regionkey =
            r_regionkey and s_nationkey = n2.n_nationkey and r_name
            = 'AMERICA' and p_type = 'ECONOMY ANODIZED
            STEEL' and
            s_acctbal :varies and l_extendedprice :varies
       ) as all_nations
group by o_year
order by o_year
```

Figure 11 : Example Query Template (QT8)

The associated plan diagram for QT8 is shown in Figure 12(a), produced by Picasso on a popular commercial database engine. In this picture, each colored region represents a specific plan, and a set of 89 different optimal plans, P1 through P89, cover the selectivity space. The value associated with each plan in the legend indicates the percentage area covered by that plan in the diagram

– for example, the biggest, P1, covers about 22% of the space, whereas the smallest, P89, is chosen in only 0.001% of the space. Compile-Time Diagrams. For instance,



(a) Plan Diagram (b) Reduced Diagram (Threshold λ = 10%)

Figure 12: Sample Plan Diagram and Reduced Plan Diagram (QT8)

the Cost Diagram quantitatively depicts the estimated query processing costs of the plans shown in the associated plan diagram, while the Cardinality Diagram displays the estimated result cardinalities. These diagrams can be drilled-down at individual locations to determine the operator trees of the plans at those locations (Schematic Plan-tree diagram), with the tree nodes optionally annotated with cost and cardinality information (Compiled Plan tree diagram).

Plan-replacement Diagrams

Perhaps the most appealing aspect of Picasso is that it also supports the construction of plan-replacement diagrams. Here, the query template's original plan/cost diagrams are taken as input, and new plan diagrams are constructed wherein a subset of the optimizer's original choices are replaced by alternative plans from the POSP set.

Run-time Diagrams

Finally, apart from the above compile-time diagrams, Picasso also generates run-time diagrams that visually describe the actual query performance behavior, in terms of execution time and result cardinalities, on the current database platform (Execution Cost and Execution Cardinality diagrams). Comparing the predicted and actual diagrams helps in understanding and profiling the modeling quality of the optimizer.

DIAGRAM PRODUCTION

The schematic architecture of the Picasso system is shown in Figure 13. The plan diagram production strategy used is the following: Given a d-dimensional query template and a plot resolution of r, the Picasso tool generates r to the power d queries that are either uniformly or exponentially (user's choice) distributed over the selectivity space. Then, for each of these query locations, based on the associated selectivity values, a query with the appropriate constants is instantiated – the constants are determined from the statistical meta-data available from the optimizer, typically in the form of histograms. This query is then submitted to the query optimizer to be "explained", that is, to have its optimal plan computed and returned.

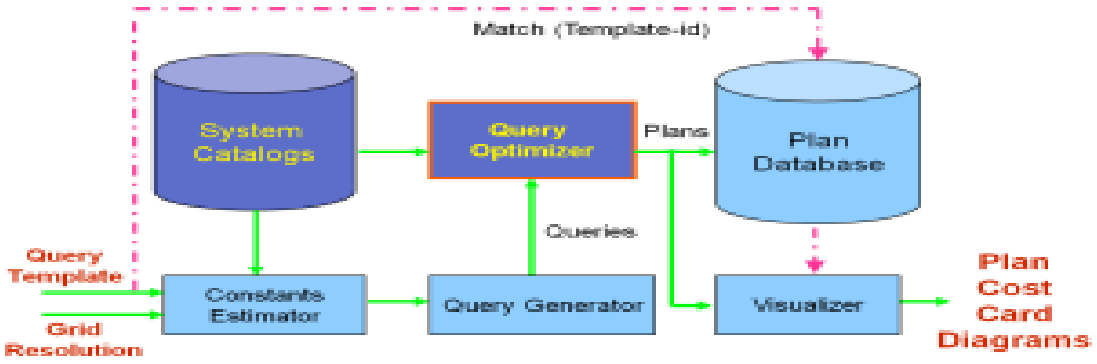


Figure 13: Picasso Architecture

After the plans corresponding to all the query points are obtained, a different color is associated with each unique plan, and all query points are colored with their associated plan colors. Then, the rest of the diagram is colored by painting the region around each point with the color corresponding to its plan.

APPLICATIONS OF PLAN DIAGRAMS

As evident from Figure 12(a), plan diagrams can be surprisingly complex and dense, with a large number of plans covering the space [3] . In fact, the very name of the Picasso tool stems from plan diagrams often appearing similar

to "cubist paintings". Plan diagrams are currently in use at various industrial and academic sites for a diverse set of applications including analysis of existing optimizer designs; visually carrying out optimizer regression testing; debugging new query processing features; comparing the behavior between successive optimizer versions; investigating the structural differences between neighboring plans in the space; evaluating the variations in the plan choices made by competing optimizers; etc.

Plan Diagrams

A plan diagram is a visual representation of the plan choices made by the optimizer over an input parameter space, whose dimensions may include database, query and system-related features. In a nutshell, plan diagrams pictorially capture the geometries of the optimality regions of the parametric optimal set of plans (POSP). To make these notions concrete, consider the parametrized SQL query template, QT8, shown in Figure 11, which is based on TPC-H Query 8. The template defines a relational selectivity space on the SUPPLIER and LINEITEM relations, with the selectivity variations specified through the s acctbal :varies and l extendedprice :varies predicates, respectively. The associated plan diagram for QT8, produced on a popular commercial database engine, is shown in Figure 12(a). In this picture, each colored region represents a specific plan, and a set of 89 different optimal plans, P1 through P89, cover the selectivity space. The value associated with each plan in the legend indicates the percentage area covered by that plan in the diagram – the biggest, P1, for example, covers about 22% of the space, whereas the smallest, P89, is chosen in only 0.001% of the space.

Applications

Plan diagrams are currently in vogue at a host of industrial and academic sites world-wide. They are employed in a diverse set of applications, including characterizing the behavior of optimizer designs; visually analyzing regression test results; investigating structural differences between neighboring plans in the parameter space; debugging new query processing features; evaluating the variations in the plan choices made by competing optimizers; etc. Apart from aiding optimizer design, plan diagrams can also be used in operational settings. Specifically, since they identify the optimal set of compile-time plans, they can be accessed at run-time to immediately identify the best plan for the current query without going through the time-consuming optimization exercise. Further, they can prove useful to adaptive plan selection techniques which, based on run-time observations, may dynamically choose to re-optimize the query and switch plans mid-way through the processing.

Different Stages

Plan Diagram Analysis

We begin by showcasing the intriguing plan diagrams obtained with popular industrial-strength optimizers in database environments based on the TPC-H and TPC-DS benchmarks. In particular, we provide compelling evidence that plan diagrams often appear similar to cubist paintings, with a large number of plans covering the space and possessing optimality regions characterized by highly intricate patterns and irregular boundaries.

Anorexic Plan Diagrams

Next, we show how complex plan diagrams can almost always be reduced to much simpler "anorexic" pictures, featuring only a few plans from the POSP set, without materially affecting the query processing quality. Our first reduction technique is based on a conservative upper-bounding of plan costs and can be applied to generic optimizers. The second, and more powerful, option leverages a "foreign plan costing" (FPC) feature now available in high-end optimizers, wherein plans can be costed outside of their native optimality regions.

Robust Plan Diagrams

We then turn our attention to the chronic problem of selectivity estimation errors faced by database systems, and explain how the plan diagram reduction scheme can be extended to identify plans that are comparatively robust to such errors. The extension is based on a generalized mathematical characterization of plan cost behavior over the

parameter space, which lends itself to efficiently establishing guarantees on the behavior of the substitute plans as compared to the optimizer's standard choices.

Plan Diagram Analysis

Skew in Plan Space Coverage

We start off our analysis of plan diagrams by investigating the skew in the space coverage of the optimal set of plans. In Table 1, we show for the various benchmark queries, three columns for each optimizer: First, the cardinality of

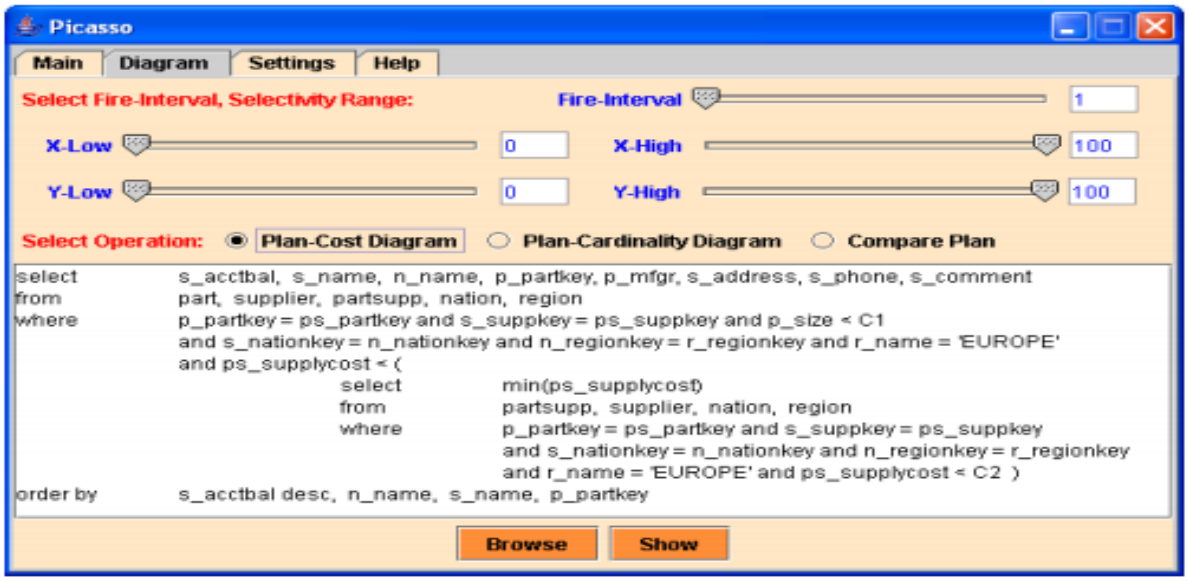


Figure 14: Picasso GUI

TPC-H Query Number	OptA Plan Card	OptA 80% Coverage	OptA Gini Index	OptB Plan Card	OptB 80% Coverage	OptB Gini Index	OptC Plan Card	OptC 80% Coverage	OptC Gini Index
2	22	18%	0.76	14	21%	0.72	35	20%	0.77
5	21	19%	0.81	14	21%	0.74	18	17%	0.81
7	13	23%	0.73	6	50%	0.46	19	15%	0.79
8	31	16%	0.81	25	25%	0.72	38	18%	0.79
9	63	9%	0.88	44	27%	0.70	41	12%	0.83
10	24	16%	0.78	9	22%	0.69	8	25%	0.75
18	5	60%	0.33	13	38%	0.57	5	20%	0.75
21	27	22%	0.74	6	17%	0.80	22	18%	0.81
Avg(dense)	**28.7**	**17%**	**0.79**	**24.5**	**23%**	**0.72**	**28.8**	**16%**	**0.8**

Table 1: Skew in Plan Space Coverage

the optimal plan set; second, the (minimum) percentage of plans required to cover 80 percent of the space; and, third, the Gini index [20], a popular measure of income inequality in economics – here we treat the space covered by each plan as its "income". Our choice of the Gini index is due to its desirable statistical properties including being Lorenzconsistent, and bounded on the closed interval, with 0 representing no skew and 1 representing extreme skew. Finally, the averages across all dense queries (10 or more plans in the plan diagram) are also given at the bottom of Table 1.

Plan Cardinality Reduction by Swallowing

Motivated by the above skewed statistics, we now look into whether it is possible to replace many of the small-sized plans by larger-sized plans in the optimal plan set, without unduly increasing the cost of the query points associated with the small plans. That is, can small plans be "completely swallowed" by their larger siblings, leading to a reduced plan set cardinality, without materially affecting the associated queries. To do this, we first fix a threshold, representing the maximum percentage cost increase that can be tolerated. Specifically, no query point in the original space should have its cost increased, post-swallowing. Next, to decide whether a plan can be swallowed, we use the following formulation:

Cost Domination Principle

Given a pair of distinct query points q1(x1; y1) and q2(x2; y2) in the two dimensional selectivity space, we say that point q2 dominates q1, symbolized by q2 Â q1, if and only if x2 ¸ x1, y2 ¸ y1, and result cardinality Rq2¸ Rq1(note that result cardinality estimations are, in principle, independent of plan choices).Then, if points q1(x1; y1) and q2(x2; y2), are associated with distinct plans P1 and P2, respectively, in the original space, C21 , the cost of executing query q1 with plan P2 is upper-bounded by C2 , the cost of executing q2 with P2, if and only if q 2 Â q1.

Intuitively, what is meant by the cost domination principle is that we expect the optimizer cost functions to be monotonically non-decreasing with increasing base relation selectivities and result cardinalities. Equivalently, a plan that processes a superset of the input, and produces a superset of the output, as compared to another plan, is estimated to be more costly to execute. Based on the above principle, when considering swallowing possibilities for a query point qs , we only look for replacements by "foreign" (i.e. belonging to a different plan) query points that are in the first quadrant relative to qs as the origin, since these points upper-bound the cost of the plan at the origin. This is made clear in Figure 15, which shows that, independent of the cost model of the dominating plan, the cost of any foreign query point in the first quadrant will be an upper bound on the cost of executing the foreign plan at the swallowed point. We now need to find the set of dominating foreign points that are within the threshold, and if such points exist, choose one replacement from among these – currently, we choose the point with the lowest cost as the replacement. Finally, an entire plan can be swallowed if and only if all its query points can be swallowed by either a single plan or a group of plans.

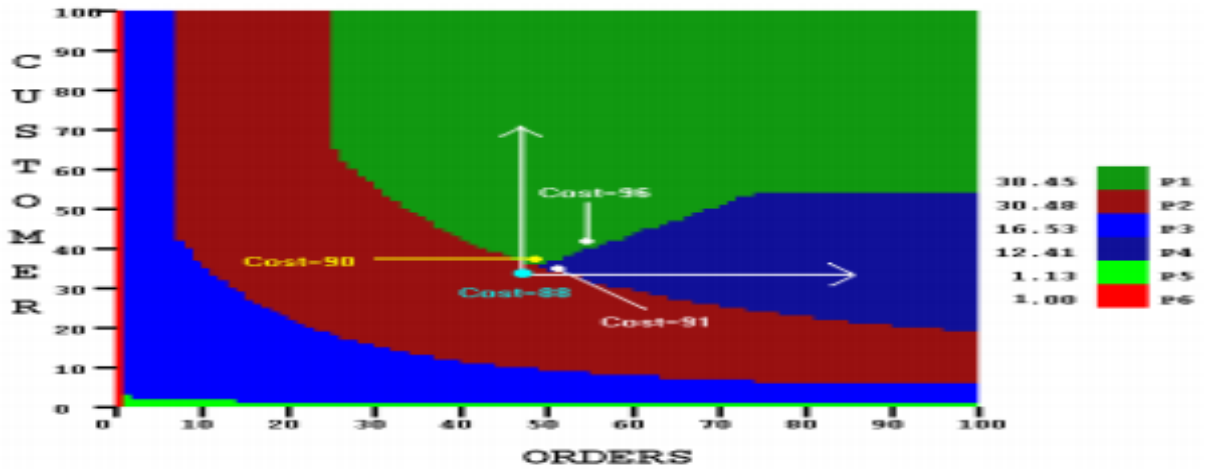


Figure 15: Dominating Quadrant

For the experiments presented here, we set , the cost increase threshold, to 10 percent. Note that in any case the cost computations made by query optimizers are themselves statistical estimates, and therefore allowing for a 10 percent "fudge factor" may be well within the bounds of the inherent error in the estimation process. When the above plan-swallowing technique is implemented on the set of plans shown in Table 1, and with ¸ = 10%, the resulting reductions (as a percentage) in the plan cardinalities are shown in Table 2. We see here that the reductions are very significant – for example, Q8 reduces by 87% (31 to 4), 84% (25 to 4) and 86% (38 to 5), for OptA, OptB and OptC, respectively. On average over dense queries, the reductions are of the order of 60% across all three optimizers, with OptC going over 70%.

Plan Reduction ! = Optimization Levels

As mentioned earlier, optimizers typically have multiple optimization levels that trade off plan quality versus optimization time, and at first glance, our plan reduction technique may appear equivalent to choosing a coarser optimization level. However, the two concepts are completely different because the optimal plan sets chosen at different levels by the optimizer may be vastly dissimilar. A striking example is Q8, where none of the 68 plans chosen by OptA at the highest level are present among the 8 plans chosen at the lowest level. Further, going to a coarser level of optimization does not necessarily result in lower plan cardinalities – a case in point is OptA on Q2, producing only 4 plans at the highest level, but as many as 22 plans at a lower level.

Relationship to PQO

The goal [21,22,23] here is to apriori identify the optimal set of plans for the entire relational selectivity space at compile time, and subsequently to use at run time the actual selectivity parameter settings to identify the best plan – the expectation is that this would be much faster than optimizing the query from scratch. Most of this work is based on assuming cost functions that would result in one or more of the following:

Plan Convexity: If a plan P is optimal at point A and at point B, then it is optimal at all points on the line joining the two points;

Plan Uniqueness: An optimal plan P appears at only one contiguous region in the entire space;

Plan Homogeneity: An optimal plan P is optimal within the entire region enclosed by its plan boundaries.

Interesting Plan Diagram Patterns

We now move on to presenting representative instances of a variety of interesting patterns that emerged in the plan diagrams across the various queries and optimizers that we evaluated in our study.

TPC-H Query Number	OptA			OptB			OptC		
	Percent Card Decrease	Average Cost Increase	Maximum Cost Increase	Percent Card Decrease	Average Cost Increase	Maximum Cost Increase	Percent Card Decrease	Average Cost Increase	Maximum Cost Increase
2	59.2	1.0	4.4	64.2	0.6	5.9	77.1	3.2	6.4
5	67.3	2.6	8.1	42.9	0.1	0.6	61.1	0.2	8.1
7	46.1	0.1	9.5	16.6	0.4	0.7	54.5	1.1	9.5
8	87.6	0.4	9.4	84	0.9	9.1	86.8	1.2	8.4
9	84.4	1.6	8.6	36.4	1.4	8.9	80.5	2.1	8.3
10	67.6	0.8	4.4	44.4	0.5	6.1	62.5	0.4	2.4
18	40.0	0.1	0.5	46.2	3.7	9.6	0	0	0
21	59.8	0.0	0.2	66.7	0.9	2.5	68.2	0.7	6.9
Avg(dense)	**67.4**	**0.9**	**6.4**	**56.9**	**0.7**	**6.1**	**71.4**	**1.4**	**7.9**

Table 2: Plan Cardinality Reduction by Swallowing

Plan Duplicates and Plan Islands

In several plan diagrams, we noticed that a given optimal plan may have duplicates in that it may appear in several different disjoint locations. Further, these duplicates may also be spatially quite separated. For example, consider the plan diagram for Q10 with OptA in Figure 16. Here, we see that plan P3 (dark pink) is present twice, being present both in the center, as well as along the right boundary of the plan space. An even more extreme example is plan P6 (dark green), which is present around the 20% and 95% markers on the CUSTOMER selectivity axis. A different kind of duplicate pattern is seen for Q5 with OptC, shown in Figure 17, where plan P7 (magenta) is present in three different locations, all within the confines of the region occupied by plan P1 (dark orange). When plans P7 and P1 are compared, we find that the former uses a nested-loops join between the small relations NATION and REGION, whereas the latter employs a sort-merge join instead.

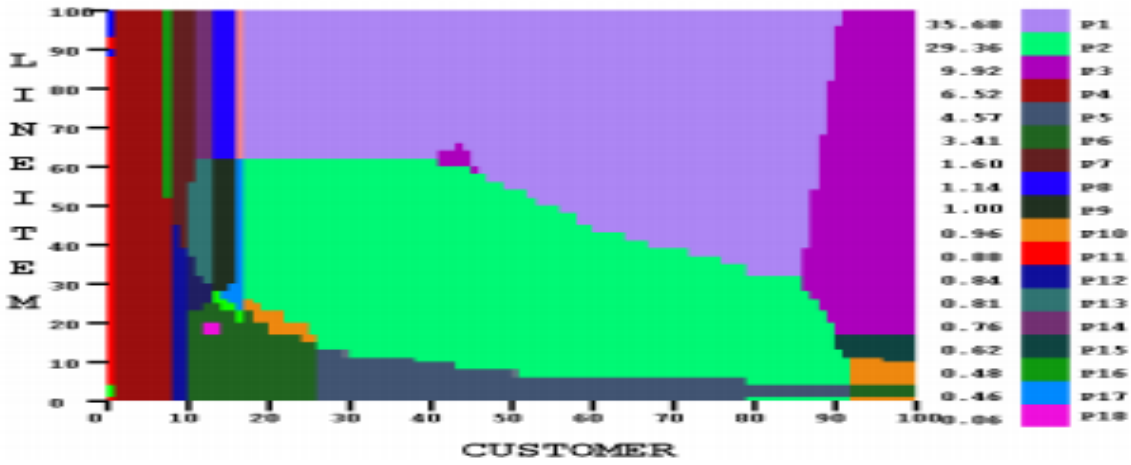


Figure 16: Duplicates and Islands (Query 10, OptA)

Apart from duplicates, we also see that there are instances of plan islands, where a plan region is completely

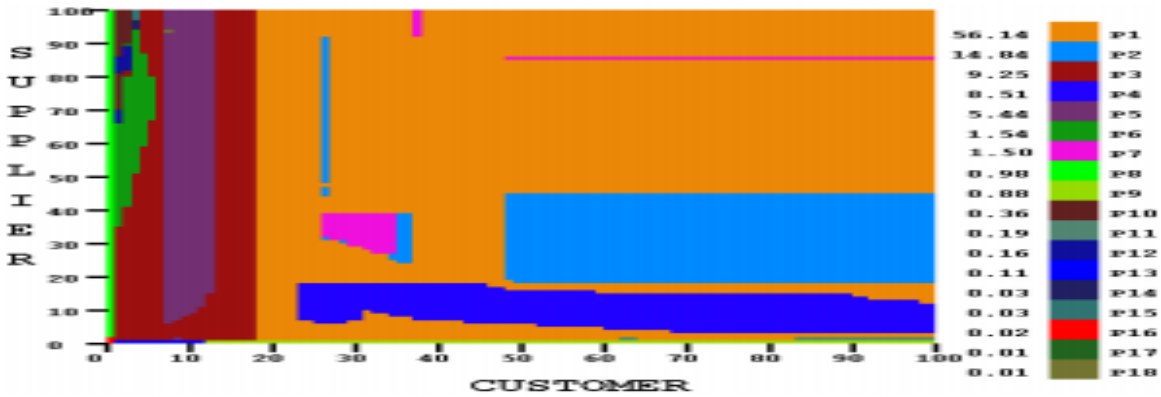


Figure 17: Duplicates and Islands (Query 5, OptC)

enclosed by another. For example, plan P18 is a (magenta) island in the optimality region of the (dark green) plan P6 in the lower left quadrant of Figure 16. Investigating the internals of these plans, we find that plan P18 has a hash join between CUSTOMER and NATION followed by a hash join with a sub-tree whose root is a nested-loop join. The only difference in plan P6 is that it first hash-joins the CUSTOMER relation with the sub-tree, and then performs the hash-join with NATION. The number of such duplicates and islands for each optimizer, over all dense queries of the benchmark, is presented in Table 3 (Original columns). We see here that all three optimizers generate a significant number of duplicates; OptA also generates a large number of islands, whereas OptB and OptC have relatively few islands

Databases	# Duplicates		# Islands	
	Original	Reduced	Original	Reduced
OptA	130	13	38	3
OptB	80	15	1	0
OptC	55	7	8	3

Table 3: Duplicates and Islands

Plan Switch Points

In several plan diagrams, we find lines parallel to the axes that run through the entire selectivity space, with a plan shift occurring for all plans bordering the line, when we move across the line. We will hereafter refer to such lines as "plan switch-points". In the plan diagram of Figure 18, obtained with Q9 on OptA, an example switch-point appears at approximately 30% selectivity of the SUPPLIER relation. Here, we found a common change in all plans across the switch-point – the hash-join sequence PARTSUPP ./ SUPPLIER ./ PART is altered to PARTSUPP ./ PART ./ SUPPLIER, suggesting an intersection of the cost function of the two sequences at this switch-point.

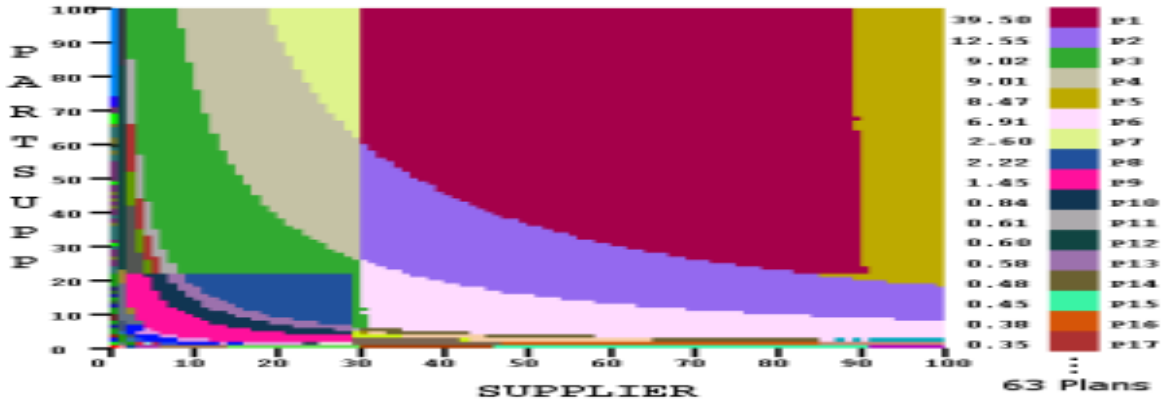


Figure 18: Plan Switch-Point (Query 9, OptA)

For the same Q9 query, an even more interesting switchpoint example is obtained with OptB, shown in Figure 19. Here we observe, between 10% and 35% on the SUPPLIER axis, six plans simultaneously changing with rapid alternations to produce a "Venetian blinds" effect. Specifically, the optimizer changes from P6 to P1, P16 to P4, P25 to P23, P7 to P18, P8 to P9, and P42 to P47, from one vertical strip to the next. The reason for this behavior is that the optimizer alternates between a left-deep hash join and a right-deep hash

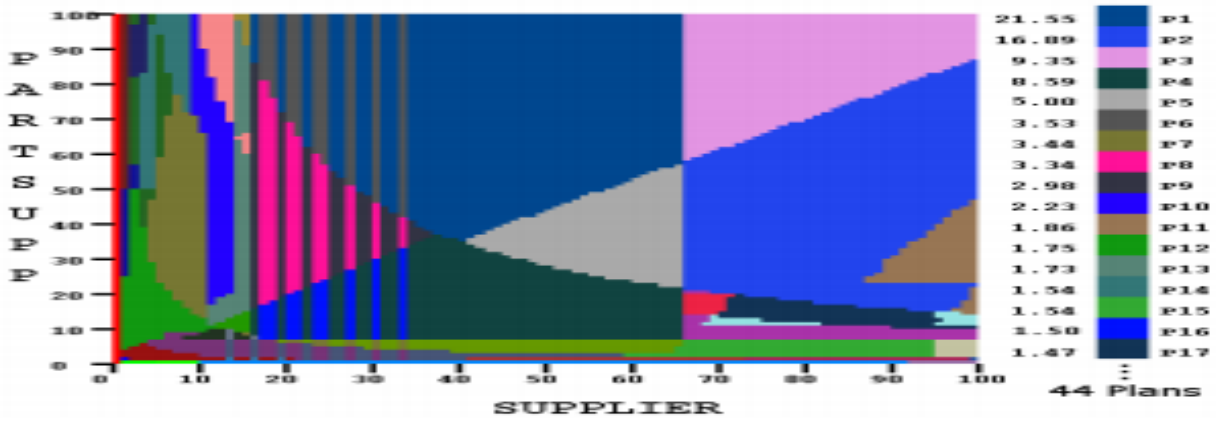


Figure 19: Venetian Blinds Pattern (Query 9, OptB)

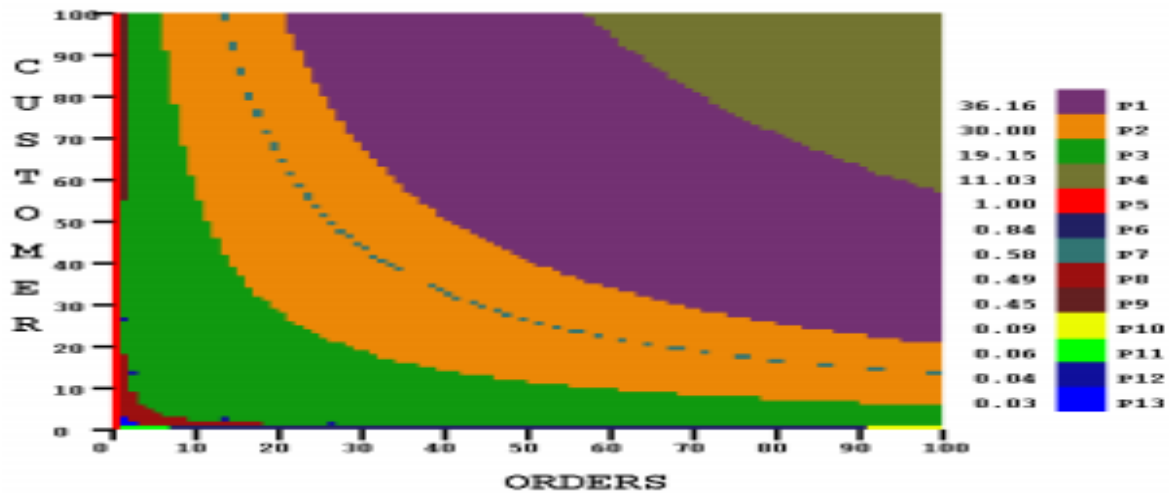


Figure 20: Footprint Pattern (Query 7, OptA)

join across the NATION, SUPPLIER and LINEITEM relations. Both variations have almost equal estimated cost, and their cost-models are perhaps discretized in a stepfunction manner, resulting in the observed blinds.

Footprint Pattern

A curious pattern, similar to footprints on the beach, shows up in Figure 20, obtained with Q7 on the OptA optimizer, where we see plan P7 exhibiting a thin (cadet-blue) broken curved pattern in the middle of plan P2's (orange) region. The reason for this behavior is that both plans are of roughly equal cost, with the difference being that in plan P2, the SUPPLIER relation participates in a sort-merge join at the top of the plan tree, whereas in P7, the hash-join operator is used instead at the same location. This is confirmed in the corresponding reduced plan diagram where the footprints disappear.

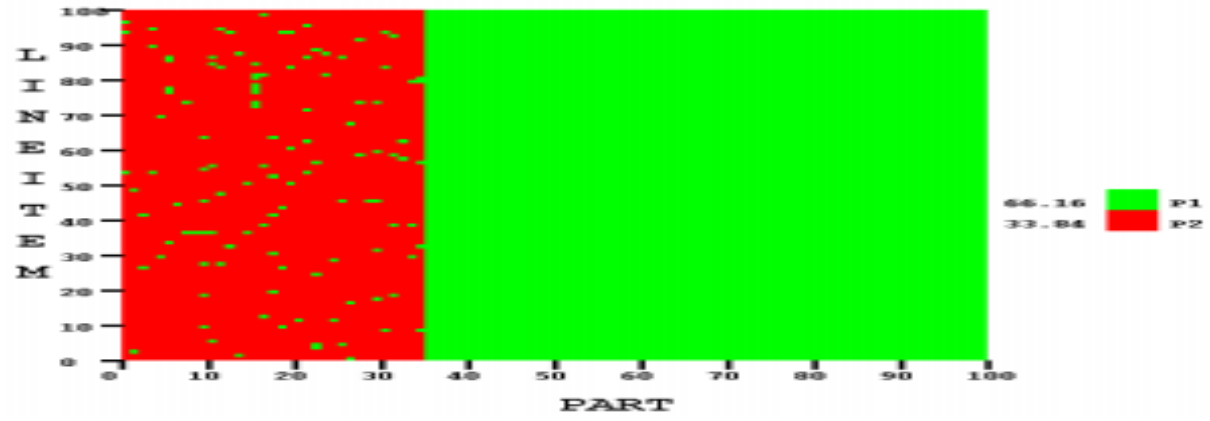


Figure 21: Speckle Pattern (Query 17, OptA)

Speckle Pattern

Operating Picasso with Q17 on OptA results in Figure 21. We see here that the entire plan diagram is divided into just two plans, P1 and P2, occupying nearly equal areas, but that plan P1 (bright green) also appears as speckles sprinkled in P2's (red) area. The only difference between the two plans is that an additional SORT operation is present in P1 on the PART relation. However, the cost of this sort is very low, and therefore we find intermixing of plans due to the close and perhaps discretized cost models.

Anorexic Plan Diagrams

with a modest cost increase [4], two thirds of the plans in a dense plan diagram are liable to be eliminated through plan swallowing. A cost increase threshold of only 20 percent is usually amply sufficient to bring down the absolute number of plans in the final reduced picture to within or around ten. Further, that this applies not just to the 2D templates considered , but also to higher-dimensional templates. In short, that plan diagrams can usually be made "anorexic" in an absolute sense while retaining acceptable query processing performance. This observation is based on our experience with a wide spectrum of dense plan diagrams ranging from tens to hundreds of plans, across the suite of industrial-strength optimizers, on TPC-H based multi-dimensional query templates. Carrying out anorexic plan reduction on dense plan diagrams has a variety of useful implications for improving both the efficiency of the optimizer and the choice of execution plan.

Contributions

We consider here the problem of reducing plan diagrams, from theoretical, statistical and empirical perspectives. We first show that finding the optimal (w.r.t. minimizing the plan cardinality) reduced plan diagram is NP-Hard through a reduction from Set Cover. This result motivates the design of CostGreedy, a greedy algorithm whose complexity is O(nm), where n is the number of plans and m is the number of query points in the diagram ($n \ll m$). Hence, for a given picture resolution, CostGreedy's performance scales linearly with the number of plans in the diagram, making it much

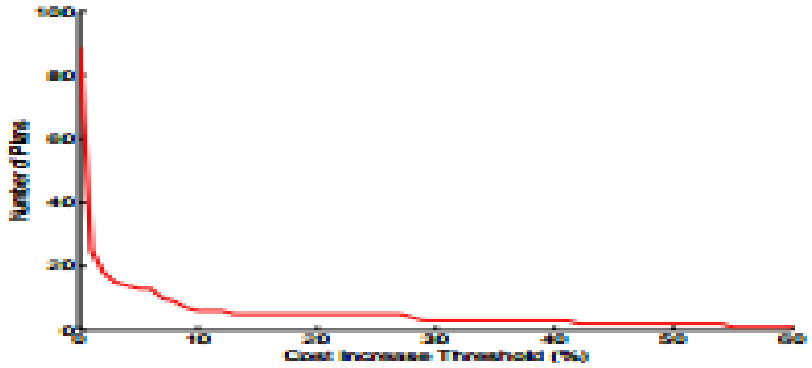


Fig 22: Plan Cardinality vs Cost Increase Threshold

more efficient than the O(m2) reduction algorithm. Further, from the reduction quality perspective, CostGreedy provides a tight performance guarantee of O(ln m), which cannot be improved upon by any alternative deterministic algorithm. We also consider a storage-constrained variant of the plan reduction problem and find that it retains the hardness of the general problem. On the positive side, however, we provide ThresholdGreedy, a greedy algorithm that delivers a performance guarantee of 0.63 w.r.t. the optimal. Using extremely coarse characterizations of the cost distributions of the optimal plans, we develop fast but effective estimators for determining the expected number of plans retained for a given threshold. These estimators can also be used to predict the location of the best possible tradeoff (i.e. the "knee") between the plan cardinality reduction and the cost increase threshold. Last, through an experimental analysis on the plan diagrams produced by industrial strength optimizers with TPC-H-based multidimensional query templates, we show that (a) plan reduction can be carried out efficiently, (b) the CostGreedy algorithm typically gives the optimal reduction or is within a few plans of the optimal, (c) the analytical estimates of the plan-reduction versus cost threshold curve are quite accurate, and finally, that (d) a 20% cost threshold is amply sufficient to bring the plan cardinality to within or around 10, even for high dimensional query templates – this is an especially promising result from a practical perspective.

ANOREXIC REDUCTION BENEFITS

The production of anorexic reduced plan diagrams, that is, diagrams whose plan cardinality is within/around a small absolute number, has a variety of useful implications for improving both the efficiency of the optimizer and the choice of execution plan:

Quantification of Redundancy in Plan Search Space

Plan reduction quantitatively indicates the extent to which current optimizers might perhaps be over-sophisticated in that they are "doing too good a job", not merited by the coarseness of the underlying cost space. This opens up the possibility of redesigning and simplifying current optimizers to directly produce reduced plan diagrams, in the process lowering the significant computational overheads of query optimization. An approach that we are investigating is based on modifying the set of sub-plans expanded in each iteration of the dynamic programming algorithm to (a) include those within the cost increase threshold relative to the cheapest sub-plan, and (b) remove, using stability estimators of the plan cost function over the selectivity space, "volatile" sub-plans; the final plan choice is the stablest within-threshold plan. The goal here is to apriori identify the optimal set of plans for the entire relational selectivity space at compile time, and subsequently to use at run time the actual selectivity parameter settings to identify the best plan.

Identification of Error-Resistant Plans

Plan reduction can help to identify plans that provide robust performance over large regions of the selectivity space. Therefore, errors in the underlying database statistics, a situation often encountered by optimizers in practice [24], may have much less impact as compared to using the fine-grained plan choices of the original plan diagram, which may have poor performance at other points in the space. In short, the final plan choices become robust to errors that lie within the optimality regions of the replacement plans.

Identification of Least-Expected-Cost Plans

When faced with unknown input parameter values, today's optimizers typically approximate the distribution of the parameter values using some representative value – for example, the mean or modal value – and then always choose this "least specific cost" plan at runtime. Computing the least expected cost plan not only involves substantial computational overhead when the number of plans is large, but also assumes that the various plans being compared are all modeled at the same level of accuracy, rarely true in practice.

Minimization of Multi-Plan Overheads

Multi-plan approaches that dynamically select the best plan at runtime by executing multiple different plans, either in parallel or sequentially[24,25] – plan reduction can help to reduce the computational overheads of these approaches by minimizing the number of alternative choices.

THE PLAN REDUCTION PROBLEM

In this section, we define the Plan Reduction Problem, hereafter referred to as PlanRed, and prove that it is NP-Hard through a reduction from the classical Set Cover Problem. For ease of exposition, we assume in the following discussion that the source SQL query template is 2-dimensional – the extension to higher dimensions is straightforward.

Preliminaries:

The input to PlanRed is a Plan Diagram, defined as follows:

Definition 1. Plan Diagram

A Plan Diagram P is a 2-dimensional [0, 100%] selectivity space S, represented by a grid of points where:

1. Each point q(x, y) in the grid corresponds to a unique query with (percentage) selectivities x, y in the X and Y dimensions, respectively.

2. Each query point q in the grid is associated with an optimal plan Pi (as determined by the optimizer), and a cost ci(q) representing the estimated effort to execute q with plan Pi.

3. Corresponding to each plan Pi is a unique color Li, which is used to color all the query points that are assigned to Pi. The set of all colors used in the plan diagram P is denoted by LP . Also, we use Pi to both denote the actual plan, as well as the set of query points for which Pi is the plan choice – the interpretation to use will be clear from the context. With the above framework, PlanRed is defined as follows:

Definition 2. PlanRed

Given an input plan diagram P, and a cost increase threshold λ (λ ≥ 0), find a reduced plan diagram R that has minimum plan cardinality, and for every plan Pi in P,

1. Pi ∈ R, or

2. ∀ query points q ∈ Pi, ∃Pj ∈ R, such that cj (q) ci (q) ≤ (1 + λ) That is, find the minimum-sized "cover" of plans that is sufficient to recolor P (using only the colors in LP) without increasing the cost of any re-colored query point (i.e. whose original plan is replaced by a sibling plan) by more than the cost increase threshold. Obviously, for λ → 0, R will be almost identical to P, whereas for λ → ∞, R will be completely covered by a single plan. In the above definition, we need to be able to evaluate cj (q), the cost of executing query point q with the substitute choice Pj . The specific bounding technique we use is based on assuming the following:

Plan Cost Monotonicity (PCM): The cost distribution of each of the plans featured in the plan diagram P is monotonically non-decreasing over the entire selectivity space S. Intuitively, what the PCM condition states is that the query execution cost of a plan is expected to increase with base relation selectivities. For most query templates, this is usually the case since an increase in selectivity corresponds to processing a larger amount of input data. In this situation, the following rule applies:

Definition 3. Cost Bounding Rule

Consider a pair of query points, q1(x1, y1) with optimal plan P1 having cost c1(q1), and q2(x2, y2) with optimal plan P2 having cost c2(q2). Then the cost of executing query q1 with plan P2, i.e. c2(q1), is upper bounded by c2(q2) if x2 ≥ x1, y2 ≥ y1. That is, when considering the recoloring possibilities for a query point q1, only those plan colors that appear in the first quadrant, relative to q1 as the origin, should be considered. The reason for restricting attention to the first quadrant is that only a vacuous statement can be made about the costs of plans from other quadrants, namely that they lie in the interval [c1(q1), ∞). Moreover, if there exists a differently colored point q2 in the first quadrant whose cost is within the λ threshold w.r.t. the optimal cost of q1, then q1 can be recolored with the color of q2 without violating the query processing quality guarantee. That is, condition 2 of Definition 2 is replaced by the stronger requirement ∀ query points q ∈ Pi, ∃Pj ∈ R, such that ∃r ∈ Pj with r in first quadrant of q and cj (r) ci (q) ≤ (1 + λ). Handling non-PCM templates. When a query template features negation operators (e.g "set difference") or short-circuit operators (e.g. "exists"), the PCM condition may not hold. However, as long as the template exhibits monotonicity (non-decreasing or non-increasing) along each of the selectivity axes, the above Cost Bounding Rule still applies with an appropriate choice of reduction quadrant, as shown in Table 4 for the 2D case.

Cost Behavior X dimension	Cost Behavior Y dimension	Reduction Quadrant
Non-decreasing	Non-decreasing	I
Non-increasing	Non-decreasing	II
Non-increasing	Non-increasing	III
Non-decreasing	Non-increasing	IV

Table 4: Reduction Quadrants

CostGreedy (Plan Diagram P, Threshold λ)

1. For each point q from $TopRight$ to $BottomLeft$ do
   (a) set $cur(q) = color(q)$
   (b) update $belong(q)$ with plans that are in q's first quadrant with cost within the given threshold
2. Let $m = m_1 \times m_2$.
3. Create n sets $S = \{S_1, S_2, \ldots S_n\}$ corresponding to the n plans.
4. Let $U = \{1, 2, \ldots m\}$ correspond to the m query points.
5. Define $\forall i = 1 \ldots n$, $S_i = \{j : i \in belong(r)$ or $i = cur(r)$ for query point r corresponding to $j, \forall j = 1 \ldots m\}$
6. Let $I = (U, S)$, I be an instance of the Set Cover problem.
7. Let L_n be the color of the $TopRight$ point. Remove set S_n and all its elements from I.
8. Apply Algorithm Greedy Setcover to I. Let C be the solution.
9. $C = C \bigcup \{S_n\}$
10. Recolor the grid with colors corresponding to the sets in C and update new costs appropriately. If a point belongs to more than one subset, use color that results in least cost increase.
11. End Algorithm CostGreedy

Fig-23: CostGreedy

GREEDY PLAN REDUCTION

We first consider AreaGreedy, where the greedy heuristic is based on plan areas. Then we present CostGreedy, a new reduction algorithm that is greedy on plan costs. Its computational efficiency and reduction quality guarantees are quantified for PlanRed. We then present ThresholdGreedy, a greedy reduction that has strong performance bounds for the storage-budgeted variant. As before, for ease of exposition, we assume that P is 2-dimensional – the algorithms can be easily generalized to higher dimensions, while the theoretical results are independent of the dimensionality.

The AreaGreedy Algorithm [4] first sorts the plans featuring in P in ascending order of their area coverage. It then iterates through this sequence, starting with the smallest-sized plan, checking in each iteration whether the current plan can be completely swallowed by the remaining plans – if it can, then all its points are recolored using the colors of the swallower plans, and these points are added to the query sets of the swallowers. By inspection, AreaGreedy clearly has a time complexity of O(m2), where m is the number of query points in P. With respect to reduction quality, let AG denote the solution obtained by AreaGreedy, and let Opt denote the optimal solution. Then, the upper bound of the approximation factor |AG| |Opt| is at least 0.5 √m.

The CostGreedy Algorithm

We propose here CostGreedy, a new greedy reduction algorithm, which provides significantly improved computational efficiency and approximation factor as compared to AreaGreedy. Consider an instance of PlanRed that has an m1 × m2 grid with

Greedy Setcover (Set Cover I)

1. Set $C = \emptyset$
2. While $U \neq \emptyset$ do:
   (a) Select set $S_j \in S$, such that $|S_j| = max(|S_i|); \forall S_i \in S$ (in case of tie, select set with smallest index)
   (b) $U = U \setminus S_j, S = S \setminus \{S_j\}$
   (c) $C = C \bigcup \{S_j\}$
3. Return C
4. End Algorithm Greedy Setcover

Fig-24: Algorithm Greedy Setcover

n plans and m = m1 × m2 query points. By scanning through the grid, we can populate the cur and belong data structures for every point. This is done as follows: For each query point q with plan Pi in the grid, set cur(q) to be i, and add to belong(q) all j such that Pj can replace q. Using this, a Set Cover instance I = (U, S) can be created with |U | = m and |S| = n. Here U will consist of elements that correspond to all the query points and S will consists of sets corresponding to the plans in the plan diagram. The elements of each set will be the set of query points that can be associated (under the λ constraint) with the plan corresponding to that set.

Complexity Analysis

In the following theorem we show that the time complexity of CostGreedy is O(nm). Since it is guaranteed that n ≤ m, and typically n ≪ m, this means that CostGreedy is significantly more efficient than AreaGreedy, whose complexity is O(m2). Further, it also means that for a given diagram resolution, the performance is linear in the number of plans in P.

Approximation Factor

We now assess the approximation factor that can always be guaranteed by CostGreedy with respect to the optimal.

LEMMA 5. CostGreedy has an approximation factor |CG||Opt|= O(ln m), where m is the number of query points in P.

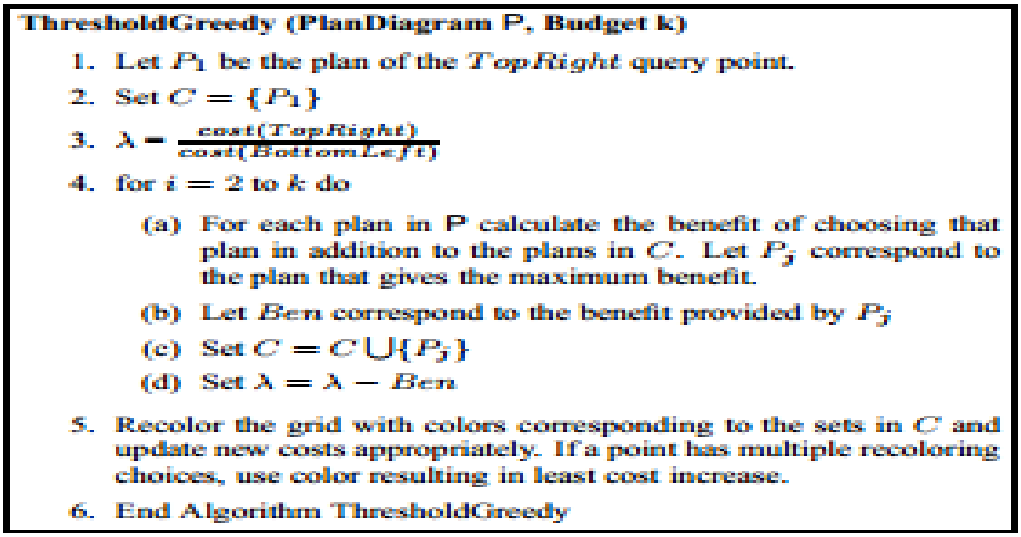
PROOF. It has been show that Algorithm Greedy Setcover (GS) has an approximation factor |GS||Opt|≤ H(m), where m is the cardinality of the universal set, and H(m) is the mth harmonic number. The input to GS can have at most (m − 1) elements in its universal set (this occurs when the TopRight query point has a unique color not shared by any other point in the entire diagram). Therefore,

|CG||Opt|=|GS||Opt|≤ H((m − 1)) = O(ln m) (1) Tightness of Bound. It is shown that given any k, l where|Greedy| = k and |Opt| = l, a Set Cover instance can be generated with (k + l) sets and m elements such that m ≥ G(k, l), where G(k, l) is a recursively defined greedy number: G(l, l) = l

G(k + 1, l) = ⌈ll − 1∗ G(k, l)⌉ It is also shown that the following tight bound of ln m for Set Cover can be achieved using such a construction when m = G(k, l): ln m − ln ln m − 0.31 ≤kl≤ ln m − ln ln m + 0.78 (2)

The ThresholdGreedy Algorithm

We now turn our attention to developing an efficient greedy algorithm for the Storage-budgeted variant of the PlanRed problem. Specifically, we present ThresholdGreedy, a greedy algorithm that selects plans based on maximizing the benefits obtained



ThresholdGreedy (PlanDiagram P, Budget k)

1. Let P_1 be the plan of the $TopRight$ query point.
2. Set $C = \{P_1\}$
3. $\lambda = \frac{cost(TopRight)}{cost(BottomLeft)}$
4. for $i = 2$ to k do
   (a) For each plan in P calculate the benefit of choosing that plan in addition to the plans in C. Let P_j correspond to the plan that gives the maximum benefit.
   (b) Let Ben correspond to the benefit provided by P_j
   (c) Set $C = C \bigcup \{P_j\}$
   (d) Set $\lambda = \lambda - Ben$
5. Recolor the grid with colors corresponding to the sets in C and update new costs appropriately. If a point has multiple recoloring choices, use color resulting in least cost increase.
6. End Algorithm ThresholdGreedy

Fig-25: Algorithm ThresholdGreedy

by choosing them. The benefit of a plan is defined to be the extent to which it decreases the cost threshold λ of R when it is chosen, which means that at each step ThresholdGreedy greedily chooses the plan whose selection minimizes the effective λ. The least number of plans that can be in R is a single plan which corresponds to the plan of the TopRight query point in P. This can be always achieved by setting the cost increase threshold λ to equal the ratio between the costs of the TopRight and BottomLeft query points in P, i.e. λSinP lan = cost(T opRight)/cost(BottomLeft). We bootstrap the selection algorithm, shown in Figure 25, by first choosing this plan and subsequently choosing additional plans based on their relative benefits. Let Benopt and Bengreedy be the total benefit of choosing k plans by the optimal and greedy algorithms, respectively. This means that the final cost increase threshold with the optimal selection is λSinP lan − BenOpt, and with the threshold greedy solution is λSinP lan − BenT G. The following theorem quantifies the approximation factor of ThresholdGreedy:

THEOREM 5. Given a storage budget of k plans, let Benopt be the benefit obtained by the optimal solution's selection, and BenT G be the benefit obtained by the ThresholdGreedy algorithm's selection. Then BenT G BenOpt ≥ 1 − (k − 1 k)k

For k = 10, which we consider to be a reasonable budget in practice, the above ratio works out to about 0.65, while for k → ∞, the ratio asymptotically goes down to 0.63. In an overall sense, this means that ThresholdGreedy is always guaranteed to provide close to two-thirds of the optimal benefit.

ESTIMATORS FOR PLAN REDUCTION

Our experience has been that CostGreedy takes only about a minute to carry out a single reduction on plan diagrams that have in the order of a million query points. While this appears sufficiently fast, it is likely that users may need to iteratively try out several reductions with different cost increase thresholds in order to identify the one appropriate for their purpose. For example, the user may wish to identify the "knee" of the tradeoff between plan cardinality

AvgEst (Plan Diagram P, Threshold λ)

1. Let $Cost(i), \forall i = 1 \ldots n$ denote the average cost of Plan P_i
2. Set $U = \{1, 2, \ldots n\}$
3. Set $S_i = \{1, 2, \ldots n\}, \forall i = 1 \ldots n$
4. for each plan P_i do
   (a) For all plans P_j such that $Cost(j) < Cost(i)$ or $Cost(j)$ is not within the threshold of $Cost(i)$, set $S_j = S_j \setminus \{i\}$
5. Apply Algorithm Greedy Setcover to I. Let C be the solution.
6. return $|C|$
7. End Algorithm AvgEst

Fig-26: Algorithm AvgEst

reduction and the cost threshold – that is, the location which gives the maximum reduction with minimum threshold. In the above situations, using the CostGreedy method repeatedly to find the desired setting may prove to be cumbersome and slow . Therefore, it would be helpful to design fast but accurate estimators that would allow users to quickly narrow down their focus to the interesting range of threshold values. AvgEst, takes as input the plan diagram P and a cost increase threshold λ, and returns the estimated number of plans in the reduced plan diagram R obtained with that threshold. It uses the average of the costs of all the query points associated with a plan, to summarize the plan's cost distribution. All these averages can be simultaneously computed with a single scan of P. AvgEst then sets up an instance of Set Cover, as shown in Figure 30, with the number of elements equal to the number of plans, and the set memberships of plans is based on their representative average costs satisfying the λ threshold. On this instance, the Greedy Set Cover algorithm, introduced earlier in Figure 28, is executed. The cardinality of the solution is returned as an estimate of the number of plans that will feature in R. Our second estimator, AmmEst, uses in addition to the average value, the minimum and maximum cost values of the query points associated with a plan. That is, each plan is effectively represented by three values. Subsequently, the algorithm is identical to AvgEst, the only change being that the check for set membership of a plan is based on not just the average value but on all three representative values (min, max and avg) satisfying the membership criterion. By iteratively running the estimator for various cost thresholds, we can quickly plot a graph of plan cardinality against threshold, and the knee of this curve can be used as the estimated knee. Our measurements show that this estimation process executes orders of magnitude faster than calculating the knee using CostGreedy.

Robust Plan Diagram

To address this problem, an obvious approach is to improve the quality of the statistical meta-data, for which several techniques have been presented in the literature ranging from improved summary structures to feedback-based adjustments to on-the-fly reoptimization of queries. Aim is to identify robust plans that are relatively less sensitive to such selectivity errors. In a nutshell, to "aim for resistance, rather than cure", by identifying plans that provide comparatively good performance over large regions of the selectivity space. These techniques provide novel and elegant formulations, but have to contend with the following issues:

Firstly, they are intrusive requiring, to varying degrees, modifications to the optimizer engine. Secondly, they require specialized information about the workload and/or the system which may not always be easy to obtain or model. Thirdly, their query capabilities may be limited compared to the original optimizer – e.g., only SPJ queries with key-based joins .Further has been implemented and evaluated on a non-commercial optimizer.

PROBLEM FRAMEWORK

For ease of exposition, we assume in the following discussion that the SQL query template is 2-dimensional in its selectivity variations – the extension to higher dimensions is straightforward.

Plan and Reduced Plan Diagrams

From a query template Q, a plan diagram P is produced on a 2-dimensional [0,1] selectivity space S by making repeated calls to the optimizer. The selectivity space is represented by a grid of points where each point q(x, y) corresponds to a unique query with selectivities x, y in the X and Y dimensions, respectively. Each q is associated with an optimal (as determined by the optimizer) plan Pi , and a cost ci (q) representing the estimated effort to execute q with plan Pi . Corresponding to each plan Pi is a unique color Li , which is used to color all the query points that are assigned to Pi .

Plan Diagram Reduction Problem: This problem is defined as follows: Given an input plan diagram P, and a maximum cost-increase threshold λ (λ ≥ 0), find a reduced plan diagram R with minimum cardinality such that for every plan Pi in P,

1. Either Pi ∈ R, or

2. ∀ q ∈ Pi , the assigned replacement plan Pj ∈ R guarantees cj (q) ci (q) ≤ (1 + λ) That is, find the maximum possible subset of the plans in P that can be completely "swallowed" by their sibling plans in the POSP set. A point worth reemphasizing here is that the threshold constraint applies on an individual query basis. For example, setting λ = 10% stipulates that the cost of each query point in the reduced diagram is within 1.1 times its original value. Therefore, an efficient heuristic-based online algorithm, called CostGreedy, was proposed and shown to deliver near-optimal "anorexic" levels of reduction, wherein the plan cardinality of the reduced diagram usually came down to around 10 or less for a λ-threshold of only 20%.

Selectivity Estimation Errors

Consider a specific query point qe, whose optimizer-estimated location in S is (xe, ye). Denote the optimizer's optimal plan choice at point qe by Poe. Due to errors in the selectivity estimates, the actual location of qe could be different at execution-time – denote this location by qa(xa, ya), and the optimizer's optimal plan choice at qa by Poa. Assume that Poe has been swallowed by a sibling plan during the reduction process and denote the replacement plan assigned to qe in R by Pre. Finally, extend the definition of query cost (which applied to the optimal plan) to have ci (t) denote the cost of an arbitrary POSP plan Pi at an arbitrary query point t in S. With respect to R, the actual query point qa will be located in one of the following disjoint regions of Pre that together cover S: Endo-optimal region of Pre: Here, qa is located in the optimality region of the replacement plan Pre, which also implies

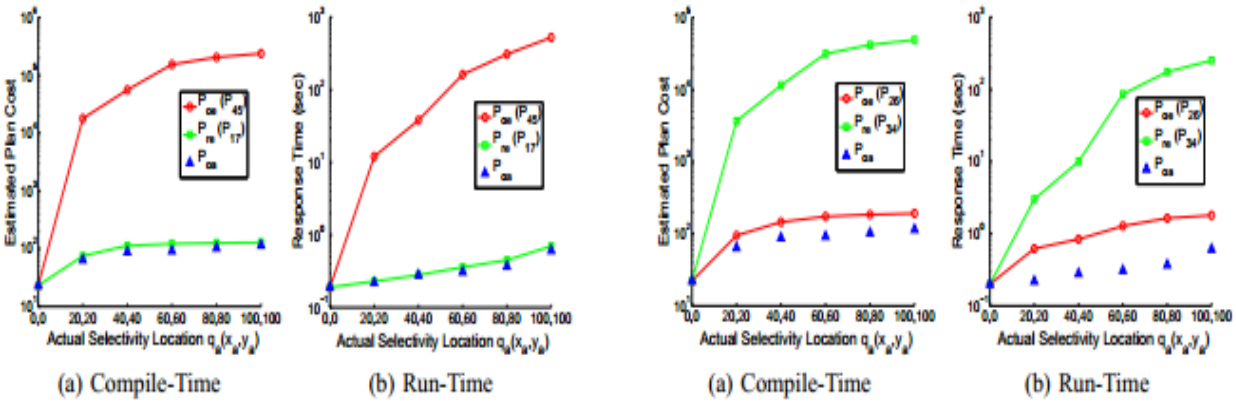


Fig-27: Beneficial Impact of Plan Replacement Fig-4: Adverse Impact of Plan Replacement

that Pre ≡ Poa. Since cre(qa) ≡ coa(qa), it follows that the cost of Pre at qa, cre(qa) < coe(qa) (by definition of a cost-based optimizer). Therefore, improved resistance to selectivity errors is always guaranteed in this region. Swallow-region of Pre: Here, qa is located in the region "swallowed" by Pre during the reduction process. Due to the λ-threshold constraint, we are assured that cre(qa) ≤ (1 + λ)coa(qa), and by implication that cre(qa) ≤ (1 + λ)coe(qa). Now, there are two possibilities: If cre(qa) < coe(qa), then the replacement plan is again guaranteed to improve the resistance to selectivity errors. On the other hand, if coe(qa) ≤ cre(qa) ≤ (1 + λ)coe(qa), the replacement is guaran-

teed to not cause any real harm, given the small values of λ . Exo-optimal region of Pre: Here, qa is located outside both the endo-optimal and swallow-regions of Pre. At such locations, we cannot apriori predict Pre's behavior, and therefore the replacement may not always be a good choice – in principle, it could be arbitrarily worse. Therefore, we would like to ensure that even if the replacement does not provide any improvement, it is at least guaranteed to not do any harm. That is, the exo-optimal region should have the same performance guarantees as the swallow-region.

Robust Reduction

From the above discussion, it is clear that we need to ensure that only safe replacements are permitted. This means that replacement should be permitted only if the λ threshold criterion is satisfied not just at the estimated point, but at all locations in the selectivity space. At the same time, it is important to ensure that the safety check is not unnecessarily conservative, preventing most plan replacements, and in the process losing all the error-resistance benefits. Therefore, the overall goal is to maximize plan diagram reduction without violating safety considerations. More formally, our problem formulation is:

Robust Reduction Problem: Given an input plan diagram P, and a maximum cost-increase-threshold λ ($\lambda \geq 0$), find a reduced plan diagram R with minimum plan cardinality such that for every plan Pi in P,

1. $P_i \in R$, or

2. $\forall q \in P_i$, the assigned replacement plan $P_j \in R$ guarantees $\forall$ query points $q' \in P$, $c_j(q') \; c_i(q') \leq (1 + \lambda)$

That is, find the minimum-sized error-resistant "cover" of plans that reduces the plan diagram P without increasing the cost of any reassigned query point by more than the cost increase threshold, irrespective of the actual location of the query at run-time.

ENSURING ROBUST REDUCTION

To find an error-resistant cover of the plan diagram, we need to evaluate the behavior of each replacement plan Pre, w.r.t. its swallowing target Poe, at all points in S. This requires, in principle, finding the costs of Poe and all potential Pre at every point in the diagram. Of course, Poe and Pre need not be costed in their respective endo-optimal regions, since these values are already known through the plan diagram production process. The remaining exo optimal costs can be obtained using the Foreign-Plan-Costing feature, hereafter referred to as FPC, that is now supported in several industrial-strength optimizers, as mentioned in the Introduction. While the above solution is conceptually feasible, it is practically unviable due to its enormous computational overheads. Plan-costing is certainly cheaper than the optimizer's standard optimal-plan-searching process [26], but the overall overhead is still O(nm) where n and m are the number of plans and the number of points, respectively, in P. Typical values of n range from the several tens to several hundreds, while m is at least in thousands, making an exhaustive approach impractical. The above situation motivates us to study whether it is possible, based on using FPC at only a few select locations, to infer the behavior in the rest of the space

Modeling Plan Cost Functions

For ease of presentation, we will initially assume that our objective is to model the cost behavior of plans with respect to a 2-D selectivity space corresponding to distinct relations Rx and Ry . In current optimizers, the operators in the execution plan are all typically either unary or binary with regard to their inputs. Therefore, given a specific plan operator tree we can define the following types of nodes:

Selectivity Nodes: These are the unary nodes that implement the selection operations on relations Rx and Ry .

Dependent Nodes: These are the nodes in the tree that have at least one Selectivity Node in the sub-tree below them.

Independent Nodes: These are all the remaining nodes in the tree that do not belong to either of the above two categories.

Node Cost Models

We now enumerate the cost models that can be associated with the above node categories on the 2-D selectivity space S. Our formulation is based on detailed observations of cost behavior of individual operators on commercial database optimizers. In the following, the variables x and y are used to denote the (fractional) selectivities on the respective dimensions.

Independent Nodes: Since these nodes do not have a Selectivity Node in their sub-tree, variations in x and y do not change their inputs, and consequently their outputs. Therefore, for a given plan, the costs at these nodes remain the same throughout S.

Selectivity Nodes: The input cardinalities for these nodes will be constant (equal to the corresponding base relation's cardinality n) while the output cardinality is directly dependent on the selectivity value. Therefore, the cost behavior can be captured by the simple linear model involving coefficients a1 and a2 shown in Table 5. For example, Table-Scans will have a1 = 0, while Index-Scans are likely to have non-zero values for both constants.

Dependent Unary Nodes: The input cardinalities for these nodes will be a function of x and/or y, and the associated family of cost models is as shown in Table 5. For operators such as Aggregates, Arithmetic Expressions, Scalar functions, etc. the simple linear model will apply, whereas the logarithmic model would apply to operators such as Sort and Group By that require multiple passes over the data.

Dependent Binary Nodes: These are the nodes that represent binary set operators such as Join, Union, Minus, etc. The different types of input possibilities and the associated cost models are shown in Table 5. Note that we deliberately do not consider the case where both the inputs to the binary node are functions of x (or y or xy).

Cost Model of a Complete Plan

The cost function of the entire plan is the aggregate sum of the costs of the individual nodes. Considering all possible cost models a node can have, we can conclude that the overall cost model of a plan for a 2D selectivity space is of the form Cost(x, y) =a1x + a2y + a3xy + a4x log x + a5y log y+ a6xy log xy + a7 (1) where a1, a2, a3, a4, a5, a6, a7 are coefficients, and x, y represent the selectivities of Rx and Ry , respectively. Modeling a specific plan requires suitably choosing the seven coefficients, and this is achieved through standard surface-fitting techniques. The extension of Equation 1 to a general d-dimensional space is straightforward, with the number of parameters in the cost model being (2 d+1 − 1) – for example, 3D cost functions are modeled using 15 parameters.

Node Type	Input Cardinalities		Cost Model
Selectivity Node ($\sigma = x$)	n		$a_1 nx + a_2$
Dependant Unary Nodes	$n_1 x$		$a_1 n_1 x + a_2$
			$a_1 n_1 x \log n_1 x + a_2$
	$n_1 xy$		$a_1 n_1 xy + a_2$
			$a_1 n_1 xy \log n_1 xy + a_2$
Dependant Binary Nodes	$n_1 x$	n_2	$a_1 n_1 x + a_2 n_2 + a_3 n_1 n_2 x + a_4$
	$n_1 xy$	n_2	$a_1 n_1 xy + a_2 n_2 + a_3 n_1 n_2 xy + a_4$
	$n_1 x$	$n_2 y$	$a_1 n_1 x + a_2 n_2 y + a_3 n_1 n_2 xy + a_4$

Table 5: Cost Models for various Node Types

Replacement Safety Conditions

For the 2D scenario, using the above 7-coefficient cost model, our goal now is to come up with an efficient mechanism to assess, given an optimal plan Poe, candidate replacement plan Pre and a cost-increase threshold λ, whether it would be safe from a global perspective to have Pre swallow Poe. Let the cost functions for Pre and Poe be fre(x, y) = a1x+a2y+a3xy+a4x log x+a5y log y+a6 xy log xy+a7 (2) and foe(x, y) = b1x+b2y +b3xy +b4x log x+b5y log y +b6xy log xy +b7 (3) respectively. Now consider the "safety function" f (x, y) = fre − (1 + λ)foe (4) which captures the differences between the costs of Pre and a λ-inflated version of Poe in the selectivity space. All points where f (x, y) ≤ 0 are referred to as SafePoints whereas points that have f (x, y) > 0 are called ViolatingPoints. For a re-

placement to be globally safe, there should be no ViolatingPoint anywhere in the selectivity space. In the following, we will use LR-Boundaries to collectively denote the left and right boundaries of the selectivity space, and TBBoundaries to collectively denote the top and bottom boundaries of the space.

For a specific value of y, the safety function f (x, y) can be rewritten as fy (x) = g1 ∗ x + g2 ∗ x log x + g3 for appropriate coefficients g1, g2, g3. Similarly, we can define fx(y). With this terminology, the following theorem provides us with conditions for checking whether the selectivity space is safe for the plan-pair (Poe,Pre) with regard to replacement.

THEOREM 1. For a plan-pair (Poe,Pre) and a selectivity space S with corners [(x1, y1), (x1, y2), (x2, y2), (x2, y1)], the replacement is safe (i.e., within λ-threshold) in S if any one of the conditions, SC1 through SC6, given in Table 2 is satisfied. The proof of the above theorem uses the following two lemmas – the first provides us with a condition that is sufficient to ensure safety of all points on the straight line segment joining a pair of safe points, while the second describes the behavior of the slope of the safety function.

LEMMA 1 (LINE SAFETY). Given a fixed y = yo, and a pair of safe points (x1, yo) and (x2, yo) with x2 > x1, the straight line joining the two points is safe if the slope f′ yo (x) is either (i) monotonically non-decreasing, OR (ii) monotonically decreasing with f′ yo (x1) ≤ 0 or f′yo (x2) ≥ 0. A similar result holds when x is fixed.

	Left Boundary	Right Boundary	Top Boundary	Bottom Boundary
SC1	Safe	Safe	$f''_{y_2}(x) \geq 0$	$f''_{y_1}(x) \geq 0$
SC2	$f'_y(x_1) \leq 0$ & Safe	Safe	$f''_{y_2}(x) < 0$	$f''_{y_1}(x) < 0$
SC3	Safe	$f'_y(x_2) \geq 0$ & Safe	$f''_{y_2}(x) < 0$	$f''_{y_1}(x) < 0$
SC4	$f''_{x_1}(y) \geq 0$	$f''_{x_2}(y) \geq 0$	Safe	Safe
SC5	$f''_{x_1}(y) < 0$	$f''_{x_2}(y) < 0$	$f'_x(y_2) \geq 0$ & Safe	Safe
SC6	$f''_{x_1}(y) < 0$	$f''_{x_2}(y) < 0$	Safe	$f'_x(y_1) \leq 0$ & Safe

Table 6: Safety Satisfaction Conditions

LEMMA 2 (SLOPE BEHAVIOR). If the slope of the safety function, f′ y (x), is non-decreasing (resp. decreasing) along the line-segments y = y1 and y = y2, then it is non-decreasing (resp. decreasing) for all line segments in the interval (y1, y2). A similar result holds for f′x (y). The test criteria of Theorem 1 are utilized for determining reduction safety in the SafetyCheck algorithm, described next. A related point to note here is that these checks are conservative in that it is possible to have global safety even if none of the conditions are met – i.e. the test is sufficient, but not necessary.

THE SEER ALGORITHM

In this section, we first describe the safety checking procedure, which given a plan-pair (Poe, Pre), responds whether the replacement of Poe by Pre is globally safe throughout the selectivity space S. We then present and analyze the SEER algorithm which uses this procedure to perform error-resistant plan diagram reduction. In the following, we will assume that the selectivity space S is represented by a grid G, with m = r × r points, i.e. the grid resolution in each dimension is r.

Safety Checking

To implement safe reduction in a 2-D plan diagram, we need to be able to check for the satisfaction of any of the conditions (SC1 through SC6) stipulated in Theorem 1. A straightforward way to achieve this is the following Perimeter Test procedure: Perimeter Test. First compute the safety function at all points on the perimeter of G – this is obtained through the foreign-plancosting (FPC) feature. Then, compute the slope behavior (nondecreasing or decreasing) along all the grid lines – this is achieved by evaluating the slopes at the matching end-points on the perimeter and comparing the values. The slope at a perimeter point is approximated by computing the value of the safety function at its immediate internal neighbor – i.e., along the "inner perimeter", and evaluating the slope of the line

segment joining these two points. Finally, use these results to verify whether any of the 6 safety conditions are satisfied. In the Perimeter test, the number of FPC operations is $2*4(r-1)$ for the perimeter (the 2 is due to having to compute both fre and foe), while the computation of the slopes takes an additional $2 * 4(r-3)$ costings of the inner perimeter, leading to a total of approximately 16r. Note that this is much less than the 2r2 FPC operations required by a brute-force approach of costing both plans at all points in the diagram. For example, with r = 100, the overhead is brought down by over an order of magnitude. An obvious minor improvement that could be carried out on the 16r overhead is to perform the inner perimeter costings only when conditions SC1 and SC4 are violated. In this case, only one of SC2 or SC3 (resp. SC5 or SC6) can be valid. Hence, we need to perform FPC operations only at two boundaries of the inner perimeter, one along each dimension. This reduces the FPC overhead to 12r. Wedge Test. We now present a powerful optimization, called Wedge Test, that allows conditions SC1 and SC4 to be checked with a constant number of FPC, specifically 24, irrespective of the resolution.

Plan Diagram Reduction

We now show how the above safety checks are integrated into the SEER procedure for plan diagram reduction. Note that SEER's design is completely different from that of CostGreedy [7] because now reduction is permitted only if it satisfies a safety criterion that is applicable over S, whereas CostGreedy's attention is limited to only Poe's endo-optimal region. The complete SEER algorithm is shown in Figure 28. Here, a Set-Cover instance is first created from the input plan diagram P. Then the two-stage global safety checking procedure of the Wedge Test, followed by the Perimeter Test, is implemented to evaluate replacement possibilities across each pair of plans in P, and the Set-Cover instance is updated accordingly. Finally, the resulting instance is solved using the standard greedy techniques to obtain the reduced plan diagram R. The above bounds and approximation factors for SEER compare

SEER (Plan Diagram P, Threshold λ)

1. Create a Set-Cover Instance $I = (U, S)$, where $S = \{S_1, S_2, ..., S_n\}$, $U = \{1, 2, ..., n\}$, corresponding to the n plans in the original plan diagram P.
2. Set each $S_i = \{i\}, \forall i = 1...n$
3. For each pair of plans (P_i, P_j) do
   if (WEDGE_TEST (P_i, P_j, λ) == Safe) then
   $S_i = S_i \bigcup \{j\}$
   else if (PERIMETER_TEST (P_i, P_j, λ) == Safe) then
   $S_i = S_i \bigcup \{j\}$
4. Solve the Set-Cover instance I using the Greedy Set-Cover algorithm to identify the plans retained in R.

Fig-28: The SEER Reduction Algorithm

very favorably with those of the CostGreedy reduction algorithm, which has time complexity O(nm) and approximation factor of O(log m), since typically n << m.

LiteSEER

A Fast Variant The SEER design makes conscious efforts, as described above, to minimize the computational overheads, but these overheads do grow with increasing dimensionality of the query template. Therefore, we have also designed and evaluated LiteSEER, a light-weight heuristic-based algorithm that trades SEER's safety guarantee for providing rapid running-times. In LiteSEER, a replacement is simply assumed to be safe if all the corner points of the selectivity space are safe. The intuition behind this observation is that when two points are safe, then the straight line joining them is also usually safe. Given a d-dimensional plan diagram P with n plans, the LiteSEER algorithm only computes the safety function at the 2d corners of the associated selectivity space. It immediately follows that its overall complexity is O(2d n +n2). LiteSEER is an optimal algorithm (complexity-wise) w.r.t. efficiency.

Future Research Directions

In the concluding part of the tutorial, we will outline a set of open technical problems and future research directions. Sample problems include the following:

a) Plan Diagram Density Classifier: Currently, only after a plan diagram is produced do we know whether it is sparse (few plans) or dense (several plans). It would be extremely useful to develop a predictor for the density of the diagram prior to production. This objective could be treated as a data mining problem involving classification, and the associated feature vector is likely to have to include aspects of the query template, the database engine, and the schematic/statistical meta-data.

b) Plan Diagram Coloring Mechanism: Currently, unique colors are assigned at random to the various plans featuring in the plan diagram. A semantically richer option would be to color plans in a manner that also reflects the extent of their structural differences. For instance, if a pair of plans happen to have the same join order, they should be assigned close shades of a common color. With this new approach to coloring, the plan diagram itself provides a first-cut reflection of the differences between plans as we traverse the selectivity space.

c) Query Execution Visualization: While plan diagrams capture the "compile-time" behavior of query optimizers, it would be instructive to also visualize the run-time behavior in a similar manner.

References:

[1] http://www.tpc.org/tpch

[2] http://www.tpc.org/tpcds

[3] http://dsl.serc.iisc.ernet.in/projects/PICASSO

[4] N. Reddy and J. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers", Proc. of 31st Intl. Conf. on Very Large Data Bases (VLDB), August 2005.

[5] Mahendra Chavan, R. Guravannavar, Karthik Ramachandra and S. Sudarshan, "DBridge: A Program Rewrite Tool for Set-Oriented Query Execution" in Intl. Conf. on Very Large Databases, 2008.

[6] A. Dey, S. Bhaumik, Harish D. and J. Haritsa, "Efficiently Approximating Query Optimizer Plan Diagrams", Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB), August 2008

[7] Harish D., P. Darera and J. Haritsa, "On the Production of Anorexic Plan Diagrams", Proc. of 33rd Intl. Conf. on Very Large Data Bases (VLDB), September 2007.

[8] Harish D., P. Darera and J. Haritsa, "Robust Plans through Plan Diagram Reduction", Tech. Rep. TR-2007-02, DSL/SERC, Indian Inst. of Science, 2007.

[9] M. Abhirama, S. Bhaumik, A. Dey, H. Shrimal and J. Haritsa, "On the Stability of Plan Costs and the Costs of Plan Stability", Proc. of 36th Intl. Conf. on Very Large Data Bases (VLDB), September 2010.

[10] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems", Proc. of ACM Symp. on Principles of Database Systems (PODS), June 1998.

[11] J. Haritsa, "The Picasso Database Query Optimizer Visualizer", Proc. of 36th Intl. Conf. on Very Large Data Bases (VLDB), September 2010.

[12] Melton, J., Simon A. Understanding The New SQL: A Complete Guide. Morgan Kaufman.

[13] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price T.G. Access Path Selection in a Relational Database System. In Readings in Database Systems. Morgan Kaufman.

[14] Ono, K., Lohman, G.M. Measuring the Complexity of Join Enumeration in Query Optimization. In Proc. of VLDB, Brisbane, 1990.

[15] Pirahesh, H., Hellerstein J.M., Hasan, W. Extensible/Rule Based Query Rewrite Optimization in Starburst. In Proc. of ACM SIGMOD 1992.

[16] Dayal, U. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers. In Proc. of VLDB, 1987.

[17] Haas, L., Freytag, J.C., Lohman, G.M., Pirahesh, H. Extensible Query Processing in Starburst. In Proc. of ACM SIGMOD, Portland, 1989

[18] "Java Database Connectivity (JDBC) API http://java.sun.com/products/jdbc/overview.html."

[19] A. Hulgeri and S. Sudarshan, "Parametric Query Optimization for Linear and Piecewise Linear Cost Functions", Proc. of 28th Intl. Conf. on Very Large Data Bases (VLDB), August 2002.

[20] http://en.wikipedia.org/wiki/Gini coefficient

[21] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis, "Parametric Query Optimization", Proc. of 18th Intl. Conf. on Very Large Data Bases (VLDB), August 1992.

[22] V. Prasad, "Parametric Query Optimization: A Geometric Approach", Master's Thesis, Dept. of Computer Science & Engineering, IIT Kanpur, April 1999.

[23] S. Rao, "Parametric Query Optimization: A Non-Geometric Approach", Master's Thesis, Dept. of Computer Science & Engineering, IIT Kanpur, March 1999.

[24] N. Kabra and D. DeWitt, "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans", Proc. of ACM SIGMOD Intl. Conf. on Management of Data, May 1998.

[25] G. Antonshenkov, "Dynamic Query Optimization in Rdb/VMS", Proc. of 9th IEEE Intl. Conf. on Data Engineering (ICDE), April 1993.

[26] A. Hulgeri and S. Sudarshan, "AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions", Proc. of 29th Intl. Conf. on Very Large Data Bases (VLDB), September 2003.