Comments to the Reviewers

We thank the reviewers for their very clear statement of the concerns they had, enabling us to work over the last few weeks to articulate better what is surprising and what is actionable.

Meta-review

- "the reviewers felt that the derived conclusions are (1) non surprising, i.e., Well known, and (2) non actionable, i.e., Nothing one can do about it."

  The last three pages (Sections 7–9) have been substantially rewritten to list explicitly the surprising results and to indicate the specific things that can (as well as one that need not) be done, deriving from the results of this research. To make space for this longer discussion, we removed about 1/2 page of methodological details that are less important.

  We know of no other paper that has presented those results or has been able to provide the rather specific guidance across DBMSes to database researchers and developers of places to improve and places where the problem is not present.

- "Furthermore, in the discussion the relationship to Haritsas work was pointed out." We now explain in Section 7 how our results go beyond Haritsa's novel work. (We mention here but not in the paper that Haritsa et al. did not look at all at query suboptimality.)

In the following, we respond to just the high-level concerns expressed in the detailed reviews, though emphasize that we have made small changes throughout the paper in response to this feedback.

Reviewer 1

- "...the results don't specify how the results can be used to improve the space of solutions in query optimization."

  Our paper is fundamentally about articulating and testing a causal model that will thus provide novel guidance to us and to the rest of the community about how to improve query optimization. This model identifies more precisely than before where more work could substantially add to the space of solutions in query optimization.

- " There isn't any discussion on how the results be integrated to improve the present DBMSes or tune the schema or queries better."

  We now discuss in some detail which research questions need more attention and which do not. We feel that the ultimate goal is to have the DBMS handle query optimization, and so do not consider manual schema or query tuning.

- "complex sub-queries ... a discussion on how the model might extend to such cases is missing."

  There are several paragraphs in Section 9, modified and expanded, which discuss extension of the model.

Reviewer 2

- "There is nothing surprising in the findings and although the methodology is interesting, it does not seem to lead to any new insight."

  See the discussion above for the changes we have made.

- "It is a bit annoying to read these conclusions without any reference to the literature where such problems are addressed."

  We have added some additional references.

  What our research does is help focus attention on more specifically where the problems still reside in query optimization. For example, even when the statistics are entirely accurate, existing query optimizers have challenges even on simple queries, a finding that this paper contributes.

- "since no attempt is made to put the results in perspective over what has been done in the past"

  We have not made such an attempt because we don't know (can't know) which research results from the past have been adequately incorporated into existing proprietary DBMSes. Our audience comprises both researchers, who can develop new solutions, and developers, who can incorporate existing and new solutions into subsequent versions of their products.

- "this could be as simple as mentioning how these problems have been addressed in the past and saying that the paper now qualitatively confirms what was informally known before."

  This is an excellent suggestion. We mention that in the fourth paragraph of Section 4 and the third bullet of Section 8. We also repeatedly discuss where *more* research is needed, so as to not imply that research hasn't been done in the past.

- "Put the result in context to what is known already and state if there are any conclusions that are different from the state-of-the-art in terms of query plans and optimizations."

  We have tried to be as clear and directed as we can, given the constraints mentioned above.

Reviewer 3

- "W1: It is not clear what is actionable about the hypotheses."

  See above about what we have added.

- "Sure, number of "effective plans", number of joins, etc will result in more complex plans, and more points where things are suboptimal, but most of these are not factors that a query optimizer can control."

  No disagreement here. These factors do have implications on challenges that query optimizers much contend with.

- "In direct contrast, reasons such as errors in statistics, independence assumptions, which can be ameliorated by better/more expensive statistics are not covered at all."

  We discussed that in our investigation, the statistics are accurate and the independence assumptions are also accurate (there is not skew in our data), and so those are not sources for the query suboptimality observed in our study. Sections 6.7 and 7 discuss the challenges to be addressed and possible engineering approaches that might help.

- "W2: ..."

  This paragraph was removed.

- "W3: The paper oversells in parts, eg the claim about manual optimization of queries in the introduction is meaningless since most queries are canned queries which are executed repeatedly."

  In our experience, canned queries are often those that are most problematic and require the most manual tuning. In any case, after clarifying that paragraph, we removed it due to space constraints.

- "W4: The paper only looks at one manifestation of suboptimality, which respect to plans that are optimal for nearby regions, i.e., only checking if the cutover point is chosen properly."

  Actually, we are not considering the specific case that the cutover point is chosen properly or improperly. Rather, change points are used in our operationalization of suboptimality, especially for proprietary systems. So change points are a fundamental notion in our methodology.

  But the reviewer is correct that our definition of suboptimality is conservative, in that we might miss some suboptimal plans. We mention that near the end of Section 5.3.

- "W5: the experiments are not on any standard benchmark."

  We don't see how that invalidates any of the findings or conclusions we report.

- "Why have you not explained what are the queries you used."

  The queries are generated randomly using the process detailed in Section 5.2.

Postscript

Our sincere thanks for the clear feedback, which has substantially benefited the paper.

A Causal Model of DBMS Suboptimality

ABSTRACT

The query optimization phase within a DBMS ostensibly finds the fastest query execution plan from a potentially large set of enumerated plans, all of which correctly compute the specified query. Infrequently the optimizer selects the wrong plan, for a variety of reasons. *Suboptimality* is indicated by the existence of a query plan that performs more efficiently than the DBMS' chosen plan, for the same query. From the engineering perspective, it is of critical importance to understand the prevalence of suboptimality and its causal factors. We study these aspects across DBMSes to identify the underlying causes. We propose a novel structural causal model to explicate the relationship between various factors in query optimization and suboptimality. Our model induces a number of specific hypothesis that were subsequently tested on multiple DBMSes, providing empirical support for this model as well as implications for fundamental improvements of these optimizers.

1. INTRODUCTION

DBMSes underlie all information systems and hence optimizing their performance is of critical importance. The DBMS's query optimizer plays an important role. But what if the optimizer *doesn't*: what if it selects the wrong plan?

This paper provides a thorough investigation into DBMS *suboptimality*, when the DBMS chooses a slower plan over a faster plan for a given query. We systematically examine the factors influencing the number of suboptimal queries. There could be multiple causes of the suboptimality. One possible cause could be some peculiarity within the tens of thousands of lines of code of that query optimizer. Another possible cause could be the query's complexity. A third possible reason could be some fundamental limitation *within the general architecture of cost-based optimization* that will always render a good number of queries suboptimal. Prior research in other domains shows that increasing the complexity negatively influences performance [3, 18].

To better understand the impact of different factors on suboptimality of query performance and the interaction between operators, especially in a dynamic environment, an experimental approach is needed. Our research introduces a novel approach to better understand the factors influencing query performance in DBMSes. Based on existing research and general knowledge of DBMSes, we propose an innovative predictive model to understand the presence and influence of suboptimality in query evaluation. We use an experimental methodology with the DBMS as a subject to test our hypotheses with respect to factors influencing suboptimality, applying to our experiment data collected over a cumulative 9 months of query executions (over 6,500 hours to run almost a million query executions). Our research falls within creative development of new evaluation methods and metrics for artifacts, which were identified as important design-science contributions [8].

The key contributions of this paper are as follows.

- We use an innovative *methodology* that treats DBMSes as experimental subjects.
- We find that for a surprisingly large portion of queries, the plan chosen by the query optimizer is not the best plan, for some cardinality, of the underlying tables.
- We propose and test a *predictive model* for DBMSes to better understand the factors causing suboptimality.
- The predictive model and these experimental results suggest several specific engineering directions.

This paper takes a scientifically rigorous approach to an area previously dominated by the engineering perspective, that of database query optimization. Our goal is to understand cost-based query optimizers as a *general* class of computational artifacts and to come up with insights and ultimately with predictive theories about how such optimizers, again, as a general class, behave. These theories can be used to further improve DBMSes through engineering efforts that benefit from the fundamental understanding that the scientific perspective can provide.

We focus here on the effectiveness of query optimization. The query optimization phase within a DBMS ostensibly finds the fastest query execution plan from a potentially large set of enumerated plans, all of which correctly compute the specified query. Occasionally the "optimizer" selects a suboptimal plan, for a variety of reasons. We define *suboptimality* as the existence of a query plan that performs more efficiently than the plan chosen by the query optimizer. From the engineering perspective, it is of critical importance to understand the phenomenon of suboptimality.

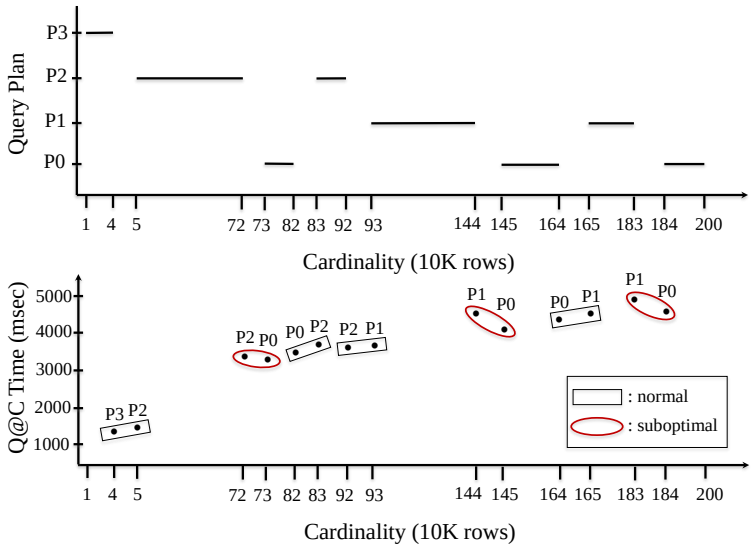

Figure 1: An Example of Suboptimality and Fluttering

We study these aspects to identify the underlying causes. We have developed an initial predictive causal model that identifies four factors that may play a role in suboptimality. The ultimate goal is to *understand* a component within a DBMS, its cost-based optimizer, through the articulation and empirical testing of a general scientific theory.

The following section introduces the methodology we will follow. We then present a predictive, causal model of suboptimality and empirically test eight specific hypotheses derived from that model. These are the first scientific results that we are aware of that test a general causal model of DBMSes. This model has implications for research in engineering more efficient DBMSes.

2. MOTIVATION

Consider a simple select-project-join (SPJ) query, with a few attributes in the SELECT clause, a few tables referenced in the FROM clause, and a few equality predicates in the WHERE clause. This query might be an excerpt from a more complex query, with the tables being intermediate results.

```
SELECT t0.id1, t0.id2, t2.idt, t1.id1
FROM ft_HT3 t2, ft_HT2 t1, ft_HT1 t0
WHERE (t2.id4=t1.id1 AND t2.id1=t0.id1)
```

The optimizer generates different plans for this query as the cardinality of the `ft_HT1` table varies, an experiment that we will elaborate later in depth.

The upper graph in Figure 1 represents the plans chosen by a common DBMS as the cardinality of `FT_HT1` decreases from 2M tuples to 10K tuples in units of 10K tuples. The x-axis depicts the estimated cardinality and the y-axis a plan chosen for an interval of cardinalities. So Plan P0 was chosen for 2M tuples, switching to Plan P1 at 1,830,000 tuples, back to Plan P0 at 1,640,000 tuples, and so on, through the sequence P0, P1, P0, P1, P2, P0, P2, and finally P3 at 40,000 tuples.

The lower graph in Figure 1 indicates the query times executed at adjacent cardinalities when the plan changed, termed the "query-at-cardinality" (Q@C) time. For some transitions, the Q@C time at the larger cardinality was also larger, as expected. But for other transitions, emphasized in red ovals, the Q@C time at the larger cardinality was smaller, such as the transition from plan P2 to P0 at 720,000 tuples. Such pairs identify suboptimal plans. For the pair at 1,440,000 tuples, P1 required 4.9sec whereas P0 at a larger cardinality required only 4.6sec. This query exhibits seven plan change points, three of which are suboptimal.

This query also illustrates an interesting phenomenon, in which the optimizer returns to an *earlier* plan. Sometimes, in fact, the optimizer starts oscillating between two plans, sometimes even switching back and forth when the cardinality estimate changes by a small percentage. The example query showed returning to P0 twice and to P1 and to P2 each once. We call this phenomenon, in which the query optimizer returns to a previous plan, "query optimizer flutter," or simply "flutter."

We have found through our experiments that flutter and suboptimality are all around us: *every* DBMS that we have examined, including several open source DBMSes and several proprietary DBMSes, covering much of the installed base worldwide, exhibit these phenomena, even when optimizing very simple queries. In the Confirmatory Experiment described in detail in Section 6.2, we started with 3056 query instances (a query run on a specific DBMS) after our extensive query measurement protocol . While about a quarter of these query instances contained only one plan, a few switched plans at almost every change in cardinality: see Figure 2. Slightly over half, 1740, exhibited suboptimality somewhere along those 200 cardinalities; a few had many changes to a plan that was in fact suboptimal, as indicated in Figure 3, across all four DBMSes considered.

One oft-stated observation is that the role of query optimization is not to get the *best* plan, but rather to get a plan that is acceptably good. Figure 4 shows the cumulative distribution of the relative amount of suboptimality (where an x-value of 100 denotes that the query ran 100% slower than the optimal query, that is, twice as long). The good news is that 1056 query executions, or 61% of the suboptimal queries, exhibited only a small degree of suboptimality: less

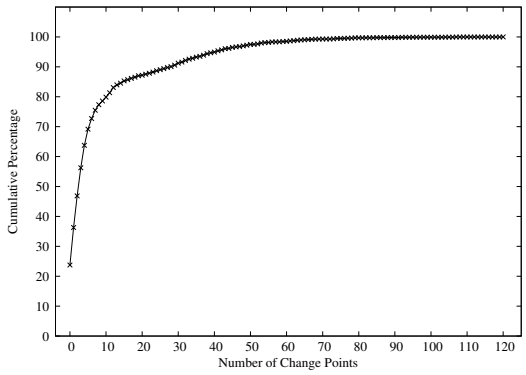

Figure 2: Cumulative percentage of queries exhibiting the indicated number of plan changes

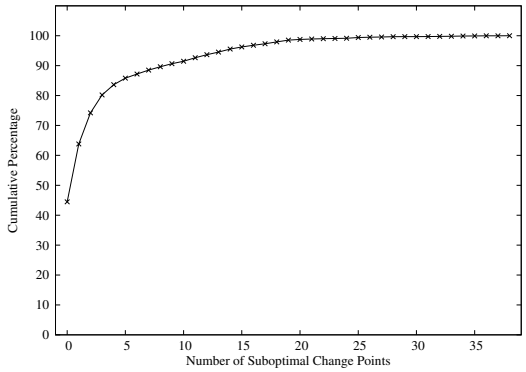

Figure 3: Cumulative percentage of queries exhibiting the indicated number of changes to a *sub-optimal* plan

than 30%. The challenge is that over fifth of all queries (684) exhibited a significant amount of suboptimality (>= 30%); the relative slowdown can be quite large for some queries.

We emphasize three important points. First, we used a sophisticated query measurement methodology that reduces the measurement jitter, so that the query plans we identify as suboptimal definitely are so. Second, these results are over four DBMSes, and thus, such phenomena are not dependent on a particular implementation of cost-based optimization. Rather, they seem to be common to *any* cost-based optimizer, independent of the specific cardinality estimation or plan costing or plan enumeration algorithm or implementation. Third, suboptimal plans are *not* the result of poor coding or of inadequate algorithms. In fact, query optimization in modern DBMSes is an engineering marvel, especially given the complexity of the SQL language and the requirements and expectations of DBMS users, who often demand that important queries simply not get slower with a new release of the DBMS. Rather, the prevalence of suboptimality observed here is a reflection of the complexity of the task of query optimization.

This is where the shift to a new methodology comes in. We want to understand cost-based optimization deeply, in this case to determine if a fundamental limitation exists, which means that we need to study multiple instances of that optimization architecture, to achieve generalizable results. To do so, we will articulate a predictive causal model

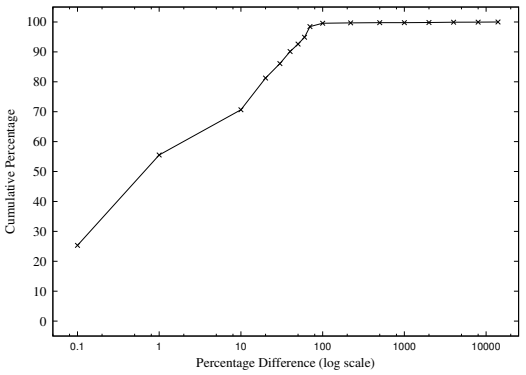

Figure 4: Cumulative percentage of queries exhibiting the indicated percentage relative suboptimality

for how and in what circumstances suboptimality arises and will provide compelling evidence via hypothesis testing that this model characterizes the behavior of query optimizers in general. Such a causal model can provide guidance to the community about where fundamental research is needed and to the DBMS engineers about where to focus their efforts.

3. RELATED WORK

SQL has emerged as the *de facto* and *de jure* standard relational database query and update language. It emerged as an ANSI and ISO standard in 1986–7, with a major extension and refinement as SQL-92 [17] and an even larger extension as SQL:1999 [16] and refinement as SQL:2008 [12] and SQL:2011.

There has been extensive work in query optimization over the last 40 years [10, 13]. Query optimization and evaluation proceeds in several general steps [19]. First, the SQL query is translated into alternative query evaluation plans based on the relational algebra via *query enumeration*. The cost of each plan is then estimated and the plan with the lowest estimated cost is chosen. These steps comprise query optimization, specifically *cost-based query optimization* [21]. The selected query plan is then evaluated by the query execution engine which implements a set of physical operators, often several for each logical operator such as join [5].

An influential survey [4] does a superb job of capturing the major themes that have pursued in the hundreds of articles published on this general topic. Chaudhuri reviews the many techniques and approaches that have been developed to represent the query plans, to enumerate equivalent query plans, to handle some of the more complex lexical constructs of SQL, to statistically summarize the base data, and to compute the cost of evaluation plans. He also mentions some of the theoretical work (which is much less prevalent) to understand the computational complexity of these algorithms. Most of this research may be classified as adopting an engineering perspective: how can we architect a query optimizer "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." [4, page 35]

To determine the best query access plan, the cost model estimates the execution time of each plan. There is a

vibrant literature on this subject [11, 15], including proposals for histograms, sampling, and parametric methods. Again, most of these papers are engineering studies, providing new techniques that improve on the state-of-the-art through increased accuracy or performance. There have also been a few mathematical results, such as "the task of estimating distinct values is *provably* error prone, i.e., for any estimation scheme, there exists a database where the error is significant" [4].

An optimizer for a language like SQL must contend with a huge search space of complex queries. Its first objective must be *correctness*: that the resulting query evaluation plan produce the correct result for the query. A secondary but clearly very important objective is *efficiency*; after all, that is the raison d'etre for this phase. As is well known, the name for this phase is an exaggeration, as existing optimizers do not produce provably optimal plans. That said, the query optimizers of prominent DBMSes generally do a superb job of producing the best query evaluation plan for most queries. This performance is the result of a fruitful collaboration between the research community and developers.

Early investigation of plan suboptimality resulted in approaches such as dynamic query-reoptimization [1, 14], which exploit more accurate runtime statistics that appear while a query is being executed, to steer in-flight plan reoptimization. The very presence of such a radical change to the normal optimize-execute sequence indicates that plan suboptimality was of interest to some researchers.

However, even with great effort over decades, optimizers as a general class are still poorly understood. As has been observed, "query optimization has acquired the dubious reputation of being something of a black art" [2]. DeWitt has gone farther, stating that "query optimizers [do] a terrible job of producing reliable, good plans [for complex queries] without a lot of hand tuning" [22, page 59]. And as we will see, suboptimality is possible even when considering only simple queries.

While this paper does not provide direct solutions to address suboptimality, we envision that by adopting our proposed predictive model for suboptimality, engineering practice, such as dynamic reoptimization just mentioned may benefit. We elaborate on this subject in Sections 6.7 and 7, where we discuss the engineering implications of our causal model.

4. A MODEL OF SUBOPTIMALITY

The purpose of query optimization is to generate optimal plans. So why would suboptimality occur in the first place? Query optimizers are highly complex, comprised of tens of thousands to even millions of lines of code. There are several reasons for this complexity. First, an optimizer must contend with the richness of the SQL language, whose definition requires about 2000 pages [12], with a multiple of linguistic features. Second, an optimizer must contend with the richness of the physical operators available to it. DBMSes have a range of algorithms available to evaluate each of many algebraic operators. Third, the optimizer must contend with an exponential number of query evaluation plans. Kabra and DeWitt [14] identify several other sources of complexity: inaccurate statistics on the underlying tables and insufficient information about the runtime system: "amount of available resources (especially memory), the load on the system, and the values of host language variables." (page 106). They also mention user-defined data types, methods, and operators allowed by the newer object-relational systems [16]. Thus, the task of optimization is very complex, with the result that the optimizers themselves consist of a collection of "components," that is, the rules or heuristics that it uses during optimization, with each of these components being itself complex.

We wish to understand the causal factors of suboptimality, through a predictive model that explicitly states the interactions between these causal factors. We test this model through experiments over thousands of queries and hundreds of thousands of query executions, showing that there is strong support for this model. We then extract engineering implications from the model, suggestions for the most productive places to look to reduce suboptimality and thus to improve existing query optimizers.

Here we examine the hypothesized influence that each independent or latent variable will have on the one dependent variable, query suboptimality. In the next section we will operationalize these variables, explaining how each is controlled or measured.

The model concerns four general constructs that we hypothesize will play a role in suboptimality: optimizer complexity, query complexity, schema complexity, and plan space complexity. Some constructs include several specific variables that contribute to that construct as a whole. Our model distills many of the widely-held assumptions about query optimization. Our contribution is the specific structure of the model and the specific operationalization of the factors included in the model.

The model depicted in Figure 5 suggests how some factors may impact the prevalence of suboptimality. This model has one dependent variable, *suboptimality*, on the far right. We observe this dependent variable in our experiments by determining whether each particular query, run on a particular DBMS and using a particular schema, is anywhere suboptimal. In Section 5, we delve into the details of how we operationalize this and other variables.

The model has several independent variables which influence suboptimality. For the construct of *optimizer complexity* we have one independent variable, "the number of operators available in the DBMS." We can manipulate this variable by our choice of DBMS: each DBMS has a set of operators available to its query evaluator and available to the query optimizer to use within query plans.

For the construct of *query complexity* we have identified two variables: "the number of correlation names in the FROM clause" and "the presence of aggregates." We have one variable for the construct of *schema complexity*: "the presence of primary keys." For each independent variables, we have interventional control in our experiments, in that we can manipulate the values of these variables, the query complexity variables through the construction of the actual query to be optimization and the schema complexity by choosing whether to associate a primary key with each of the tables.

In the middle of the figure is the construct of *plan space complexity*. Given a particular query, its measurable complexity will impact the total number of candidate plans considered by the optimizer. However, this latter factor is not directly observable, again, especially for proprietary systems. However, we *can* measure the number of plans actually generated by the optimizer when presented with different cardinalities of the underlying tables. We term this set of plans

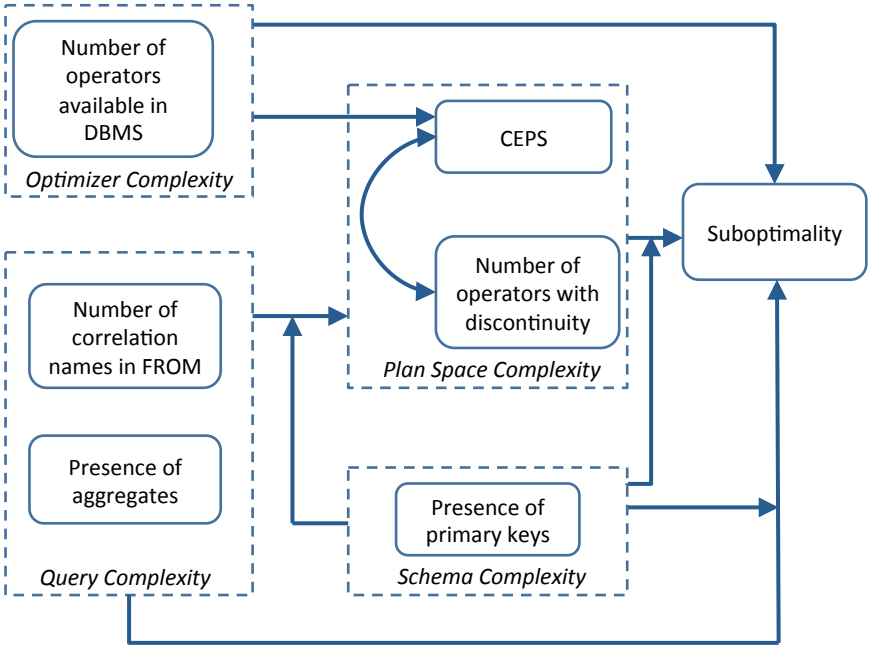

Figure 5: Predictive Model of Suboptimality

the "effective plan space" and the number of such plans, the "cardinality of the effective plan space," or *CEPS*. Similarly, we cannot directly observe the cost model utilized by the optimizer for the operators it considers, but can classify the cost model of each operator as continuous (smoothly varying with cardinality of its input(s) and without jumps) or discontinuous (sometimes producing observable jumps in the predicted cost), and can thus count "the number of operators with discontinuity." (Such operators have also been termed "nonlinear" [9], though it is the more specific aspect of discontinuity that we focus on.)

Plan space complexity is a *latent construct*, in that it is dependent on some of the variables on its left and exerts influence on the dependent variable on its right. We can observe these variables within experiments but cannot directly intervene. For example, we can influence the CEPS through manipulating the values for the independent variables of optimizer and query complexity, but cannot directly specify a value for CEPS within an experiment.

Given these four constructs and seven specific variables depicted in Figure 5, let's now examine the causal relationships between these variables. Such relationships are specific *interactions* between the the constructs (that is, their associated variables), as hypothesized by this predictive causal model.

One causal factor of this model is the optimizer complexity. We hypothesize that optimizer complexity has influence over suboptimality both directly, as depicted by the top line, and indirectly, via plan space complexity. Focusing on the direct interaction, we specifically hypothesize the following.

Hypothesis 1: As the number of operators increases, suboptimality will increase.

Focusing on the other interaction with optimizer complexity, we hypothesize that an optimizer with a larger number of available operators will generate more plans.

Hypothesis 2: As the number of operators increases, CEPS will increase.

We now turn to query complexity, a construct associated with two independent variables. As with the optimization complexity construct, we include in our model two interactions, one direct to suboptimality and one indirect through plan space complexity.

A high value of each of these specific variables implies a more complex query, which will involve more components within the query optimizer.

Hypothesis 3: As the number of correlation names in the FROM clause increases, suboptimality will increase. This will hold also, though more weakly, for the presence of aggregates,

Also, we hypothesize that the query complexity variables will influence suboptimality indirectly by influencing plan space complexity. The number of correlation names in the FROM clause will influence the number of plans considered, which will increase both variables in plan space complexity. A similar analysis holds for the presence of aggregates.

Hypothesis 4: The number of correlation names will positively influence plan space complexity: CEPS and number of operators with discontinuity.

Hypothesis 5: The presence of aggregates will weakly positively influence plan space complexity: CEPS and number of operators with discontinuity.

Let's now turn to plan space complexity. We hypothesize a positive correlation between the two variables associated with this construct. As the number of plans considered by the optimizer increases, so should CEPS, which could increase the number of operators with discontinuity that is observed.

Hypothesis 6: Plan space complexity (CEPS) and number of operators with discontinuity will be positively correlated.

We hypothesize that greater plan space complexity should make it more difficult to optimize the query. It is important to note that the occurrences of sub-optimality is not necessarily dependent on the presence of a discontinuous plan operator. However, we predict that when discontinuous plan operators appear in candidate plans, they may introduce complexity to query optimization, especially at the cardinality where discontinuity is possible. This is because the performance of such operators is sensitive to the input size in a rather complex way.

Hypothesis 7: Plan space complexity, that is, CEPS and number of operators with discontinuity, will positively influence suboptimality.

The schema complexity construct consists of the variable "presence of primary keys." Primary keys provide an integrity constraint that the optimizer can use; consideration of this constraint enables the optimizer to possibly do a better job, while also adding complexity to the optimization process. We hypothesize that this factor has a more complex role in the model, serving as a *moderator* of two interactions previously introduced. We hypothesize that the overall effect of the primary key moderator is to reduce the strength of the interaction between number of correlation names and CEPS and between presence of aggregates and CEPS.

Hypothesis 8: The presence of primary keys will negatively moderate the strength of interactions between the query complexity construct, that is, number of correlation names in FROM and presence of aggregates (weakly), and the plan space complexity construct, that is CEPS and number of operators with discontinuity, on suboptimality.

We have just *described* how suboptimality might arise, through a theory and its elaborated causal model. We now need to move to *prediction*. How might we test such a model?

The first step to test this model is to *operationalize* each variable. In the next section we describe explicitly how each variable is defined. For the independent variables, we must be able to intervene, that is, set their values before the experimental test commences. For the latent and dependent variables, we need to be able to measure their values during each experiment.

5. VARIABLE OPERATIONALIZATION

In this section we specify how we operationalized each of the seven variables in the model. Recall that each variable is a property of a DBMS (*number of operators*), of the schema (*presence of primary keys*), of the plan space (*CEPS* and *number of discontinuous operators*), or of a query (*number of correlation names* and *presence of aggregates*). The one dependent variable of our model is *suboptimality*.

5.1 Schema Complexity

The presence of primary keys is easiest to operationalize. We generate two databases, one without any primary keys and one with a primary key specified for each table on the first attribute.

5.2 Query Complexity

Query complexity is also easy to operationalize. We randomly generate queries, such as the example presented in Section 2. Each query is a select-project-join-aggregate query, with a few attributes in the SELECT clause, a few tables referenced in the FROM clause, a few equality predicates in the WHERE clause, and zero or more aggregate functions. As such, some of the complexities mentioned by Kabra and DeWitt [14], such as user-defined data types, methods, and operators, are not considered. The queries were generated by a simple algorithm.

The SELECT clause will contain from one to four attributes. The number of correlation names in the FROM clause varied from one to four, with duplication of tables allowed (duplicate table names within a FROM clause implies a self-join). In the queries that were generated, from one to four unique tables were mentioned in the FROM clause. Somewhat fewer tables were mentioned than the number of correlation names, as the presence of self-joins reduced the number of unique tables referenced by the queries.

For the query in Section 2, the number of correlation names is 3 and the presence of aggregates is false.

We ensure that Cartesian products are eliminated. We do this by connecting all the correlation names that appear in the FROM statement via equi-joins on random attributes. The comparisons are all equality operators. To ensure that the queries are as simple as possible, we do not include any additional predicates in the WHERE clause. Also for simplicity, we include neither disjunctions nor negations.

5.3 Suboptimality

We now consider the one dependent variable. How might suboptimality be observed? We have developed a system, DBLab, that allows us to perform experiments to study this phenomenon of suboptimality. DBLab submits queries to the DBMS, while varying the cardinality of one of the tables, requesting in each case the evaluation plan chosen by the DBMS. This is done using the `EXPLAIN` SQL facility available in modern DBMSes. (The Picasso system also used this facility to visualize the plan space chosen by a DBMS optimizer [6, 7].) We can then compare the performance (execution time) of various plans for the query, to identify those situations when a suboptimal plan was chosen, when in fact there was a different plan that was semantically equivalent to the chosen plan (that is, yielded the identical result) but which ran faster.

It is important to note that all manipulation must be done *outside* the DBMS. For proprietary systems we do not have access to the internal code. We do not know (*cannot* know) all the plans that were considered, nor the details of how the plans are selected. But such access is not needed; indeed, to be able to study a phenomenon across many DBMSes, such access is not practical. But by designing the experiment to examine the plans that each DBMS actually produces, and thus to examine phenomena that can be externally visible, we can obtain valuable insights into general classes of computational artifacts. Our ultimate goal is to make statements that hold across cost-based optimizers in general.

Our definition of suboptimality assumes that actual execution time for any query plan is *monotonic non-decreasing*, that is, unchanged or increasing as the cardinality increases. The intuitive justification is that at the higher cardinality, the plan *has* to do more work, in terms of CPU time and/or I/O time, to process the greater number of tuples. (For an operator having a discontinuous cost model, as the anticipated input cardinality grows, there will be *jumps*, in which the predicted cost is temporarily much greater. Section 5.5 explains further how we specifically operationalize this property. Also note that we don't consider SQL operators such as `EXCEPT` that are not monotonic.) We formalize this property as follows.

Definition: *Strict Monotonicity*: given a query Q and an actual cardinality a,

$$\forall p \in plans(p)\ \forall c > a\ (time(p,c) \geq time(p,a))$$

where $plans(p)$ is the set of plans generated by the optimizer and $time(p,\ c)$ is the execution time of plan p on the data set with the varying table at cardinality c. ■

Note that the comparison is with the same plan p, occurring at higher cardinalities.

To test this assumption, we ran an experiment that we term "Monotonicity." This experiment considered 60 queries, chosen from the pool of queries generated for testing suboptimality, and timed them for cardinalities from 10K to 2M tuples, in steps of 10K tuples (hence, we used 200 cardinalities), for each DBMS (for one DBMS that was very slow, we started with 30K tuples, as discussed in Section 6.2).

Assuming a normal distribution for our time measurements, 95% of the distribution falls between $-2 * \sigma$ and $+2 * \sigma$. Therefore, to statistically infer with a 95% confidence interval that a violation occurred, we relax our definition of monotonicity below.

Definition: *Non-Strict Monotonicity*: given a query Q, an actual cardinality C, and the standard deviation of the query executions for cardinality C as σ, Q is non-strict monotonic if $\forall p \in plans(p)\ \forall c' > C$

$$time(p,c') \times (1 + \sigma_{c'}) \geq time(p,C) \times (1 - \sigma_C)\ .$$ ■

We observed only 9,718 violations (0.61%) of non-strict monotonicity, across all the DBMSes we studied, justifying our conclusion that in fact the DBMSes under study are monotonic.

We can now turn to suboptimality. Recall that the monotonicity test examines two adjacent Q@Cs for which the *same plan* is observed. To detect suboptimality, we look for adjacent Q@Cs with *different plans*, termed a "change point," mentioned earlier. We look for such change points where the computed query time at the *upper* cardinality is *smaller* than the computed query time at the *lower* cardinality. Say the lower cardinality used Plan A and the upper cardinality exhibited Plan B. Had the DBMS query optimizer selected Plan B for the lower cardinality, the query time would have been smaller than that for Plan A, which follows directly from the monotonicity assumption. The conclusion is that for the lower cardinality, the optimizer picked the less efficient plan, and thus, this query exhibits suboptimality. Note that since this approach cannot consider plans that were never chosen, it very likely misses some suboptimal plans (for which there was a better plan not seen), and thus produces a conservative estimate of suboptimality.

Our definition of suboptimality compares the computed runtimes and standard deviations at the cardinality just before the change point $(n-1)$ and at the change point (n).

The query is said to be suboptimal if

$$time_{n-1} - 0.5 \cdot stddev_{n-1} \geq time_n + 0.5 \cdot stddev_n\ .$$

Suboptimality is coded as four levels (0–3), based on the distance in half standard deviations, up to 1.5 standard deviations. We then sum this value over all pairs of cardinalities with different plans (the change points) for the query.

Note that while DBLAB examines the plan at every cardinality, it only has to actually execute the query at pairs of cardinality that are change points. Since many fewer Q@Cs were involved, we could try many more queries than the Exhaustive Experiment. In the Exploratory Experiment to be described in Section 6.2, the value of suboptimality ranged from 0 (no suboptimality) to 78, with the majority between 0 and 9. A full 51% of the queries were suboptimal. Because the occurrence of large values of this measure was so rare, we did a log transformation: $\log_{10}(1 + subopt)$ in the confirmatory analysis.

5.4 Optimizer Complexity

By "number of operators in the DBMS" we mean the number of operators available for selection, projection, join and aggregate functions. Our experiments intervene on this variable by selecting a particular DBMS on which to evaluate each query. Across the available DBMSes, as the number of operators available increases, the complexity of the optimizer increases because it has to choose between more operators.

The `EXPLAIN PLAN` facility specifies the operators(s) employed in that plan. For each DBMS, we collect all the unique operators from all the plans and count the number of these distinct operators, each used by for least one query at at least one cardinality by that DBMS. The number of operators used by a DBMS ranged from 5 to 15 across all the queries and datasets used in the Exhaustive, Exhaustive with Keys, and Exploratory Experiments, to be discussed in Section 6.2.

5.5 Plan Space Complexity

This latent construct includes two variables.

As discussed in Section 4, the "cardinality of the effective plan space" (CEPS) is the number of plans selected as optimal for that query being evaluated on one of the 200 cardinalities for the variable table. It is "effective" because it was chosen, as opposed to all of the plans that were considered but not chosen. (Recall that for proprietary DBMSes, we do not have access to such plans.) Note that we count only distinct plans. As we saw in Figure 1, fluttering queries return to a previous plan. This particular query has a CEPS of 5.

We also wanted to evaluate the contribution of cost model non-linearity to suboptimality. We found that it is possible to assemble, from outside the DBMS, an approximation of the cost formula for that operator, and thus directly observe whether it is non-linear. Specifically, the result of SQL's `EXPLAIN PLAN` (a result that is particular to each DBMS) includes the *estimated cost* of each stage of the plan. That cost (estimate) is a dependent variable, one that can be observed.

To operationalize the "number of operators with discontinuity" we classify each DBMS operator as either continuous

or discontinuous, to be defined shortly. We then examine the queries to identify the plan change points, which provide the *effective plan space*, a set of plans chosen for one or more cardinalities of the variable table. For each distinct query plan in the effective plan space, we count the number of discontinuous plan operators that appear in that plan. Some of these plans may contribute no such operators, some may contribute one such operator, and some may contribute several such operators. We then sum the counts of the distinct plans in the effective plan space, yielding an integer as the operationalization of "number of discontinuous operators" for each query. This count per query varied from 0 to 82 for the queries we tested in the confirmatory analysis.

To classify each operator as continuous or discontinuous, we use the data from the Exhaustive Experiment, described in Section 5.3. For our study, we used a small subset of queries to be used later in testing the model, to attempt to observe the same operators as encountered in that study. We collected all query plans at all possible cardinalities (200 in all) and looked for "jumps," that is, when the cost model for an operator in the plan exhibited discontinuity. Jumps are determined from the estimated cost extracted from each plan operator. We provide an example of a jump below. If a jump is observed, we classify that particular plan operator as discontinuous.

A *jump* is a pair of close cardinalities (in our case, separated by 10K tuples) in which the query optimizer's cost model is *discontinuous*, that is, does not smoothly increase from the lower cardinality to the upper cardinality. To identify jumps in the first step, we examine the slopes between each pair of consecutive cardinalities (again, separated by 10K tuples). We expect that the slope (that is, the first derivative) is well-behaved and thus does not change much for consecutive cardinalities. A jump thus indicates a large deviation in the second derivative of the cost model across the two cardinalities.

The Exhaustive experiment gathers the cost model for each operator in each plan for each cardinality (in our case, for cardinalities ranging from 10K to 2M in steps of 10K, or 200 points). Figure 6 presents for a single query, the *cost* of each of the hash-join operators utilized in each plan at a particular cardinality. Each distinct point type depicts an individual hash-join operator. Note that when two identical query plans appear at two different cardinalities, we consider all the plan operators found at the same position within these two plans to be identical. This figure shows only a small portion of the 200 cardinalities, and is typical (we generally observed jumps at low cardinalities for these particular queries and relation sizes). As shown by this figure, the hash-join operator represented by '+' has two discontinuous jumps, both appearing below cardinality 150K. The other hash-join operator represented by $\times$ has one discontinuous jump. Between the jumps, the behavior is linear. It is our guess that such a jump is caused by the transition between one pass of a disk-based hashing technique to two passes (and indeed for one of the operators, perhaps the one lower in the operator tree, the transition to three passes, hence, the two jumps). That said, all that matters for our predictive model is that operators that experience such discontinuity in their cost models present opportunities for suboptimality.

To identify such jumps, we examine the second derivative

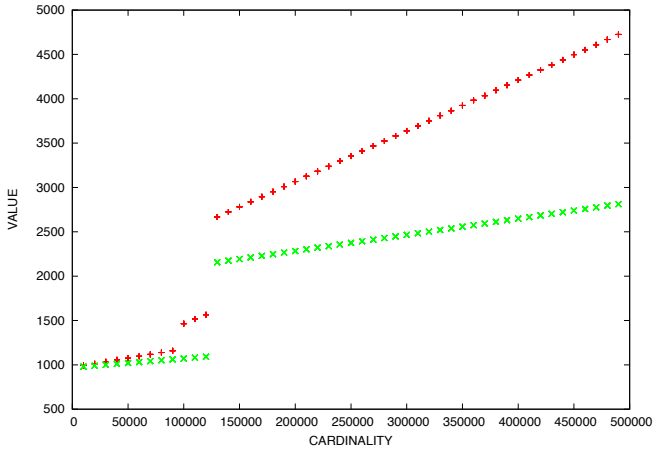

Figure 6: An Example of Discontinuous Plan Operators (Hash-Join)

(the change in the first derivative, that is, the change in the slope). A jump will be indicated by a larger than normal second derivative. By identifying a sudden change in the second derivative, we can effectively spot the cardinality at which an operator in a plan becomes discontinuous.

Formally, we compute the slope (S) of the *estimated* cost (of the cost model, formalized as C_{card}, where *card* is the input cardinality) between each pair of consecutive cardinalities as $S_{card} = (C_{card+10K} - C_{card})/10K$. This computes a series of slope values. We then compute the standard deviation of all the slope values. For example, examining Figure 6, the slopes are small except at three places, one for the green operator and two for the red operator. By identifying the slope values that are greater than one standard deviation over the average value, the discontinuous operators can be identified: A "discontinuous operator" is one for which a jump is observed in that operator in one of the plans for at least one of the input queries.

6. TESTING THE CAUSAL MODEL

In the following, we describe in detail the data used by our experiments and the experimental scenarios we defined.

6.1 Datasets

We generate our experiment dataset randomly in the experiments. However, we use seeds to control the random data generator so that it can produce repeatable data as required. (Randomly generated data will probably be easier for the optimizer to deal with than skewed data.)

Our experiment dataset consists of relational tables. There are two types of tables. The first is a "fixed table" that, once created and populated, will never be modified in the future. In contrast, the second type is a "variable table." We alter the cardinality of this table as the experiments are being performed.

We used a dataset with four tables, each with four integer-typed attributes. Each table is populated with one million rows. We also produced a version of the dataset with primary keys (of the first attribute) for all tables.

6.2 The Experiments

We are interested in predicting the behavior of DBMSes through our model. We selected four relational DBMSes, some open source and some proprietary, that are representative of the relational DBMS market. Each was used in its stock configuration.

In each experiment, we varied the cardinality from 2M (maximum) to 10K (minimum), in increments of 10K. For the one DBMS that was slower than the others and was timing out for the majority of the queries when run between 10K and 2M, we reduced the size of the tables and varied the cardinality from 60K (maximum) to 300 (minimum), in increments of 300.

We performed five separate experiments, each looking at a different aspect. It is important to emphasize that while the *queries* all came from the same query pool and while the data sets were also shared by the experiments, the *query executions* for the five experiments are disjoint. As a side comment, we mention that for all four DBMSes, the query plans generated by a DBMS for a particular Q@C of a particular query varied between the experiments, but not between the query executions (QEs) of that query instance at that cardinality, by virtue of the way we designed the measurement protocol.

The first experiment, termed "Monotonicity," was described in Section 5.3. This experiment ran quickly, as it only involved 12,000 QEs. That experiment helped us realize that we needed to be much more sophisticated in our approach to timing queries.

After we developed our six-step measurement protocol, we performed our second experiment, "Exhaustive," which more accurately tested the monotonicity assumption, as also described in Section 5.3. This experiment involved 160 query instances, 32,000 Q@Cs, and thus 320,000 QEs, requiring 1,672 cumulative hours. The (very small) percentage of strict monotonicity violations observed was consistent with the remaining variance of the query time measurement, concluding that none of the DBMSes violated monotonicity. This also provides a validation of our definition of suboptimality, which requires monotonicity.

We used the plans generated from Exhaustive to classify operators as continuous or discontinuous, as discussed in detail in Section 5.5. Again we note that this experiment used a subset of the set of *queries* (but *not* query executions) from the query pool, to ensure that we see the same operators as encountered in that study.

We also ran an experiment on the Exhaustive query set but using the data set with primary keys, termed "Exhaustive with Keys." However, in this case we did not actually execute the queries, but rather just examined the plans that were returned from the DBMS to identify additional discontinuous operators.

The fourth experiment was for exploratory analysis of the causal model, run on a representative sample of queries and data sets: (a) 600 queries of one DBMS, (b) 120 queries of another DBMS, (c) 10 queries from each other two DBMSes, all without primary keys defined, and (d) 10 queries from each of the four DBMSes on the primary key dataset, for exploration across all combinations, a total of 780 query instances. These query instances are also a subset of those used in the third experiment. We termed this experiment "Exploratory." We ran the DBMSes on the *change points*: consecutive cardinalities (10K apart; 300 for one DBMS) with different plans. This experiment required 560 hours. This experiment involved some 8,842 such change points and thus required 88,420 QEs, with the experimental methodology retaining 8,171 Q@Cs (7.6% were dropped), covering 683 query instances. Only 166 strict monotonicity violations (0.36%) and 24 relaxed ones (0.05%) were observed. This exploratory analysis allowed us to refine the operationalizations.

The fifth and final experiment was used for confirmatory analysis of the model and thus was called "Confirmatory." This was the most time-consuming of the experiments. Here we used (a) 800 queries on the data set without primary keys defined and (b) 390 of those queries that had joins on the primary key attributes, on the data set with primary keys, for a total of 1,190 queries and a total of 4,760 query instances (= 1,190 × 4 (DBMSes)). Again, we ran the DBMSes at the 57,978 change points that were observed, roughly 12 per query. This necessitated 579,780 query executions, requiring a cumulative 4,310 hours, or 6 months, with the methodology accurately timing some 44,608 Q@Cs.

In the remainder of this section, we focus using the measured independent, latent, and dependent variables in the Confirmatory experiment to test predictions that arise out of our causal model in Figure 5.

6.3 Descriptive Statistics

Several initial conclusions can be drawn from this confirmatory experiment, which was the result of several years of programming effort and many months of experimental runs. First, *every* DBMS exhibits suboptimality. Thus, this phenomenon is likely to be a fundamental aspect of either the algorithm (cost-based optimization) or the creator of the algorithm (human information processing). Our model includes both effects. In fact, a little over half (1579 queries out of the 3056 query instances that emerged from the measurement protocol) exhibited suboptimality somewhere in the range of cardinality of the varying table. Most instances of suboptimality (1237) had at least one instance of suboptimality at level 3. CEPS ranged from 1 to 41, with many queries having up to 14 plans across the cardinality range. The number of discontinuous operators observed in plans for a query averaged 9.9, with 82 being the maximum.

6.4 Correlational Analysis

We tested Hypotheses 1–7 using the strength and significance of correlations of variables involved. These hypotheses predicts eleven positive relationships. Table 1 lists the hypotheses followed by the correlation observed when testing each hypothesis. ("NS" denotes not significant at the 0.05 level, the accepted standard for significance. "—" denotes no prediction arising from the model.) As can be seen, most of the predictions (nine) arising from the causal model are supported and significant. The two exceptions are Hypotheses 1 and 2. Hypothesis 1 was not supported because the correlation between number of operators and suboptimality was negative, while we predicted positive correlation. Hypothesis 2 was not significant.

6.5 Regression Analysis

We ran a regression over the independent variables of the model that predict suboptimality over the data from the Confirmatory experiment. Our model explained 34% of the variance of the suboptimality dependent variable.

Variable	*Suboptimality*	*CEPS*	*Discontinuity*
Operators in DBMS	**H1:**−0.16	**H2:** NS	—
Correlation names	**H3:** 0.32	**H4:** 0.54	**H4:** 0.49
No. of aggregates	**H3:** 0.02	**H5:** 0.10	**H5:** 0.33
CEPS	**H7:** 0.48	—	**H6:** 0.66
Discontinuity	**H7:** 0.31	—	—

Table 1: Testing Hypotheses 1–7: Correlations

Variable	*Suboptimality*		*CEPS*		*Discontinuity*	
	Not PK	PK	Not PK	PK	Not PK	PK
Correlation names	0.44	0.41	0.56	0.52	0.53	0.41
No. of aggregates	0.01	0.02	0.07	0.15	0.32	0.35
CEPS	0.54	0.51	—	—	—	—
Discontinuity	0.41	0.40	—	—	—	—

Table 2: Testing Hypothesis 8: Interaction Strength

We also did regressions on the causal variables for CEPS and for number of operators with discontinuity. Our model explained 31% of the variance for CEPS and 35% of the variance for discontinuity.

Hypothesis 8 predicts that the presence of primary keys will be a moderator, affecting the interactions between query complexity and plan space complexity, and between query complexity and suboptimality. It thus provides predictions that the strength of such interactions will decrease in the presence of primary keys. When primary keys were not specified, our model explains 36% of the variance of suboptimality, 33% of the variance of CEPS, and 40% of the variance of discontinuity, with all of the independent variables. When primary keys were defined on the underlying tables, our model explains 30% of the variance of suboptimality, 28% of the variance of CEPS, and 27% of the variance of discontinuity. These findings are all consistent with our hypothesis predicting a negative moderation effect of primary keys.

We also examined the individual contributions to each of these three variables (e.g., suboptimality) from their causal variables (in the case of suboptimality, from the four variables of correlation names, number of aggregates, CEPS, and discontinuity). Table 2 shows that the strength of all interactions goes down as predicted, except for ones involving number of aggregates. Recall that the predictive model indicates weaker effects generally for aggregates, as they are evaluated late in the query, and the primary key attributes are not necessarily included in the grouping attributes.

6.6 Summary of Model Testing

In our experimental design, we started with a structural causal model that encapsulates our theory for how cost-based query optimizers might select a suboptimal plan for a query at a cardinality. This model implies the eight specific hypotheses listed in Section 4. We then performed four experiments to refine our operationalizations (such as discontinuity, via the Exhaustive Experiment and number of operators with the additional Exhaustive with Keys Experiment), to test fundamental assumptions (such as monotonicity, via the Monotonicity Experiment), and to test and make minor refinements to our model (via the Exploratory Experiment). While exploring the data, one must be cognizant of the possibility of Type 1 errors: false positives that lead one to believe a relationship exists when it doesn't.

To control for Type 1 errors, we then performed the Confirmatory Experiment on a completely different data set consisting of many more query instances, 4,760 in all, running on four DBMSes that each utilize cost-based query optimization, to test our refined model. Statistical inference is only possible in confirmatory analysis, where the model and hypotheses are selected a priori.

Correlational analysis provides significant support for the model, except for Hypotheses 1 and 2, with the implication that the influence of the number of operators in a DBMS remains unexplored. Via regression analysis, the model explains 34% of the variance of suboptimality (overall: with primary keys a little lower and without primary keys a little higher due to that moderating effect), 31% of CEPS, and 35% of discontinuity. In all but one case, the causal influence of independent and latent variables is significant ($p < 0.05$). The direction of the regression coefficients, again in all but one case as predicted, also provides strong support for the model.

6.7 Identifying Root Causes of Suboptimality

Our goal in this paper has been to understand cost-based query optimizers as a *general* class of computational artifacts and to articulate and test a predictive model characterizing how such optimizers, again, as a general class, behave. This model can be used to further improve DBMSes through engineering efforts that benefit from the fundamental understanding that the scientific perspective can provide.

Our model includes three causal factors of suboptimality: DBMS optimizer complexity, query complexity, and plan space complexity. Of these factors, the regression coefficient that was highest was for CEPS (cf. Table 2: 0.51–0.54, normalized) The next highest regression factor was number of tables involved, which is highly correlated with CEPS (cf. Table 1: 0.54). These two observations imply that the number of plans being considered is a major determinate of suboptimality.

The next most influential factor is the number of discontinuous operators (cf. Table 2: 0.40–0.41, normalized), implicating the cost model.

These factors implicate two root causes of suboptimality across DBMSes: (i) the cost model and (ii) the plan search process.

7. ENGINEERING IMPLICATIONS

We studied a particular phenomenon, that of the optimizer selecting a wrong plan. From the engineering perspective, it is of critical importance to understand the prevalence of suboptimality and its causal factors.

Through a series of experiments managed by DBLab, we uncovered several surprising results that provide systematic clues as to where current optimizers come up short and how they can be further improved.

- For many queries, a majority of the ones we considered, the optimizer picked the wrong plan for at least one cardinality, even when the cardinality estimates were completely accurate and even for our rather simple queries.

- A quarter of the queries exhibited significant suboptimality (>= 20% of the runtime) at some cardinality.

These two results indicate that there is still research needed on this topic. Fortunately, the causal model helps point out specifically where that research should be focused.

- Some queries exhibited significant *query thrashing*, with a plan change at almost every cardinality. While this phenomenon was first visualized by Haritsa et al. [6, 7] on some complex queries, we have shown that it is present even in a surprising percentage of simple queries.
- Furthermore, some queries exhibited many changes *to a suboptimal plan* as the cardinality was varied.

These particular queries, as well as those of the righthand side of Figure 4 exhibiting a large degree of suboptimality, can be a starting point for identifying the root cause(s) of query thrashing. The phenomenon can be investigated initially on a per-DBMS basis. Our methodology could then be used to test proposed causal mechanisms of query thrashing across DBMSes, to ascertain the generality of any proposed solutions.

- The causal model and our experimental results suggest that more research is needed to improve the cost model of *discontinuous operators*.
- We also show that it may well be useful to explicitly take *cardinality estimate uncertainty* into account.
- This research indicates that aggregates are *not* a problem, so that aspect of query optimization is in good shape.

We see that costing of discontinuous operators is a root cause of query suboptimality, and is thus particularly challenging to a query optimizer. If the cost model is even a little bit off, the optimizer might be on the wrong side of the "jump," thereby selecting the wrong plan. That the presence of discontinuous operators has such a high regression coefficient provides a quite specific guideline: more research is needed to improve the accuracy of the cost model for such operators, such as careful calibration that tunes the cost model with more accurate resource knowledge, including the global memory capacity available, as well as to improve the algorithms that allocate those buffer pages to specific operators.

Concerning the plan search process, another identified root cause of suboptimality, in cases where the DBMS is not as sure about the cardinalities of the underlying relations or the speed of the disk (e.g., if such relations migrated frequently [20]), perhaps the optimizer should explicitly take uncertainty into account. Indeed, others have started to argue that uncertainties in the query planning process should be acknowledged and exploited [2].

We mentioned dynamic query optimization earlier. Dynamic query-reoptimization normally requires a significant amount of information to be recorded during query execution, which can incur non-negligible overhead on the overall query performance [1, 14]. We envision that by utilizing the proposed predictive model for suboptimality, it may be possible to enhance reoptimization techniques such that given a particular query, a particular data distribution, and a specific plan operator, just the important statistics that affect the operator's performance can be identified and should be recorded, thereby reducing the overhead of bookkeeping irrelevant information.

Hence, the methodology introduced in this paper suggests fairly specifically where additional engineering is needed (the cost model of discontinuous operators and accommodating cardinality estimate uncertainty) and is not needed (costing of aggregation).

8. SUMMARY

This paper studies an important component of a DBMS, the query optimizer. This component is an amazingly sophisticated piece of code, but is still not well understood after decades of research and development.

This paper makes the following contributions in an attempt to gain new understanding of this component.

- Shows that even for simple queries, the prevalence of *query suboptimality* and *query thrashing*, two problems that have not been systematically investigated across DBMSes, is high, and thus there is still research needed on this mature topic.
- Introduces a new *methodological perspective* that treats DBMSes as experimental subjects.
- Proposes *operationalizations* of several relevant measures that apply even to proprietary DBMSes, as well as an overarching *predictive model* that attempts a causal explanation of suboptimality, encoding some of what is known about query optimization.
- Tests *eight hypotheses* deductively derived from the causal model. A correlational analysis and a regression analysis provided *strong support* for our model, across DBMSes, thus qualitatively confirming what was informally known.
- Uncovers compelling *evidence* (a) that suboptimality correlates with two operationalizations of query complexity, (b) that suboptimality correlates with two operationalizations of plan space complexity, (c) that query complexity is a contributor to plan space complexity, and (d) that schema complexity, as operationalized by the presence of primary keys, moderates these three interactions.
- For the kinds of queries we looked at, the factors that we identified in our model: optimizer complexity, query complexity, and plan space complexity, in concert predicted a significant portion of the variance of suboptimality. It is doubtful that any other factor, as yet unknown, will itself predict as much variance as the factors we studied in this paper. That said, it is certain that there remain several unknown causal factors; identifying those factors will undoubtedly also have important engineering implications.
- Provides a *path toward scientific progress* in the understanding of a key enabling technology. It is important to emphasize that our model doesn't apply to just one implementation of the algorithm or to one DBMS. Rather, it is quite broad, applying to any DBMS with a cost-based optimizer.
- Thereby identifies *specific directions for engineering interventions*.

This paper thus suggests a framework of model elaboration and directed engineering efforts that could reduce the prevalence of query suboptimality to an acceptably low level.

9. FUTURE WORK

There are several directions we wish to take this work. With the model refinements we propose here, additional directed pointers to engineering efforts should emerge.

We want to look into query thrashing in greater detail, as that phenomenon provides a concrete indicator of problematic optimizer behavior.

These investigations were based on very simple SPJ (select-project-join) queries. We wish to consider *schema complexity*: foreign keys and indexes, *query complexity*: complex predicates, subqueries, and user-defined data types, methods, and operators, and *instance complexity*: skewed data. Our causal model is extensible, in that we can add other factors, as long as their proper operationalization can be established, and additional causal links. So for sub-queries, we can add nesting level, type of sub-query (scalar, correlated, etc.), and number of "effective joins" (as some queries can be rewritten into joins with the outer level). It would also be interesting to study how a sub-optimal subquery can affect the suboptimality of the containing query. For instance, is it true that if many subqueries are themselves suboptimal, does that causally impact whether the overall query is suboptimal? Finally, we may be able to determine whether certain query rewrite techniques are employed within each DBMS, introducing another "DBMS complexity" factor into our model.

Kabra and DeWitt have identified another source of complexity: inaccurate statistics on the underlying tables and insufficient information about the runtime system: "amount of available resources (especially memory), the load on the system, and the values of host language variables." [14, p. 106]. Might there be other unanticipated interactions, that are unknown simply because they haven't been looked for?

By employing an extensible causal model, many complex factors can be studied via a systematic, statistically sound, scientific manner to better understand the causal factors and their interactions.

In addition to the refinements of the causal model just discussed, our research has provided specific directions for implementation interventions: refining the cost model of discontinuous operators, improving buffer allocation algorithms, and accommodating cardinality estimate uncertainty.

The methodology introduced in this paper and the causal model that results has suggested both the scope of the problem of query suboptimality and a number of specific engineering efforts that can now be carried out. Our ultimate goal is a refined causal model that can fully explain how query suboptimality arises in cost-based optimizers, thereby enabling engineering solutions that reduce and eventually effectively eliminate query suboptimality.

10. REFERENCES

[1] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing", in *Proceedings of the ACM SIGMOD Conference*, pp. 261–272, 2000.

[2] B. Babcock and S. Chaudhuri, "Towards a Robust Query Optimizer: A Principled and Practical Approach," in *Proceedings of the ACM SIGMOD Conference*, pp. 119–130, Baltimore, Maryland, 2005.

[3] D. J. Campbell, "Task Complexity: A Review and Analysis," *Academy of Management*, 13(1), pp. 40–52, 1988.

[4] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems," in *Proceedings of the ACM PODS Conference*, pp. 34–43, Seattle, WA, 1998.

[5] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys* 25(2), pp. 73–170, June 1993.

[6] D. Harish, P. Darera, and J. R. Haritsa, "On the Production of Anorexic Plan Diagrams," in *Proceedings of the VLDB Conference*, pp. 1081–1092, 2007.

[7] J. R. Haritsa, "The Picasso Database Query Optimizer Visualizer," *PVLDB* 3(2):1517–1520, 2010.

[8] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly* 28(1):75–105, 2004.

[9] A. Hulgen and S. Sudarshan, "AniPQO: almost non-intrusive query optimization for nonlinear cost functions," in *Proceedings of the VLDB Conference*, pp. 766-777, 2003,

[10] Y. Ioannidis, "Query Optimization," *ACM Computing Surveys* 23(1):121–123, June 1996.

[11] Y. Ioannidis, "The History of Histograms (abridged)", in *Proceedings of the VLDB Conference*, pp. 19–30, September 2003.

[12] ISO, "ISO SQL:2008 International Standard," 2008.

[13] M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys* 16(2):111–152, June 1984.

[14] N. Kabra and D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," in *Proceedings of the ACM SIGMOD Conference*, pp. 106–117, 1998.

[15] M. V. Mannino, P. Chu, and T. Sagar, "Statistical Profile Estimation in Database Systems," *ACM Computing Surveys*, 20(3), pp. 192–221, 1988.

[16] J. Melton, **Advanced SQL:1999**, Morgan Kaufmann, 2003.

[17] J. Melton and A. R. Simon, **Understanding the New SQL: A Complete Guide**, Morgan Kaufmann, 1993.

[18] D. L. Moody, "Metrics for Evaluating the Quality of Entity Relationship Models," *Proceedings of International Conference on Conceptual Modeling*, pp. 211–225, Springer, Singapore, 1998.

[19] R. Ramakrishnan and J. Gehrke, **Database Management Systems**, Third Edition, 2003.

[20] F. R. Reiss and T. Kanungo, "A Characterization of the Sensitivity of Query Optimization to Storage Access Cost Parameters," in *Proceedings of the ACM SIGMOD Conference*, pp. 385–396, 2003.

[21] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lori, and T. G. Price, "Access Path Selection in a Relational Database System," in *Proceedings of the ACM SIGMOD Conference*, pp. 23–34, 1979.

[22] M. Winslett, "David DeWitt Speaks Out," *ACM SIGMOD Record* 31(2), pp. 50–62, June 2002.