A Causal Model of DBMS Suboptimality

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 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's chosen plan, for the same query. From an engineering perspective, it is of critical importance to understand the prevalence of suboptimality and its causal factors. We propose a novel structural causal model to explicate the relationship between various factors in query optimization and suboptimality. Our model associates suboptimality with the factors of complexity of the query and optimizer and concomitant unanticipated interactions among the components of the optimizer. This model induces a number of specific hypothesis that were subsequently tested on multiple DBMSes. We observe that **TODO:** *Rick: check these* suboptimality prevalence correlates positively with the number of operators available in the DBMS (one proxy for optimizer complexity) and with the number of plans generated by the optimizer (another proxy for optimizer complexity) and with the number of correlation names (a measure of query complexity), providing empirical support for this model as well as implications for fundamental improvements of these optimizers. The results of this study also uncovered evidence of a potential fundamental limitation of cost-based optimization. **TODO:** *Rick:be more specific! Couple with contributions and conclusions* This paper thus provides a new methodology to study mature query optimizers, that of empirical generalization, proposes a novel causal model for query suboptimality, and tests several hypotheses deriving from this causal model.

ACM Reference Format:
A Causal Model of DBMS Suboptimality *ACM Trans. Datab. Syst.* 1, 1, Article 1 (February 2016), 41 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

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. Prior research in other domains shows that increasing complexity negatively influences performance [3; 21]. A third possible reason could be some fundamental limitation *within the general architecture of cost-based optimization* that will always render a number of queries suboptimal.

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 better understand the presence and influence of suboptimality in query evaluation. We use an experimental methodology on a collection of DBMSes as subjects to test our hypotheses with respect to factors influencing suboptimality, utilizing experiment data collected over a cumulative 16,000 hours (over two years) of query executions. Our research falls within creative development of new evaluation methods and metrics for artifacts, which were identified as important design-science contributions [10].

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 a *predictive model* for DBMSes to better understand the factors causing suboptimality.
— We test the seven quite specific hypotheses that arise from that model across a wide range of DBMSes, queries, and data.
— We show that the hypothesized interactions are **TODO:** *Rick:check after Sabah* supported, lending credibility to our particular model.
— Through an innovative analysis, we track the net cumulative benefit of a succession of operators added to the DBMS, and show that current relational DBMSes may have already reached the stage where it is difficult to provide performance improvement via new operators, and in fact performance may degrade.
— The predictive model and these experimental results suggest several specific engineering directions.
— A subsequent analysis implies that a new approach, fundamentally different from that utilized over the paper forty years, may be required to get past a fundamental limitation uncovered in this research.

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.

One might ask, shouldn't the task of optimizing queries be left to DBAs? In databases, and especially in data warehouses, the number of users writing and running queries has been growing exponentially. This growth is aided, in part, by the drag and drop query tools provided by the different systems. For example, subject areas allow business users to write queries without knowing anything about the underlying database structure. This, coupled with constantly growing data presents new challenges for tuning. For example, a large data warehouse we are familiar with runs about 30,000 queries on a daily basis. Also, tuning a subject area or tables for one group of queries can negatively impact the performance of other queries. Query optimization experts often take hours to tune and test existing canned queries; the amount expended on this one system for manual query optimization approaches $100K/year. Therefore, we argue that it is important to understand how existing query optimizers can be further improved, and indeed, whether fundamental limitation inherent in these optimizers exist and whether such limitations are indeed already being encountered.

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.

We study these aspects across DBMSes to identify the underlying causes. We have developed a predictive causal model that identifies four factors that may play a role

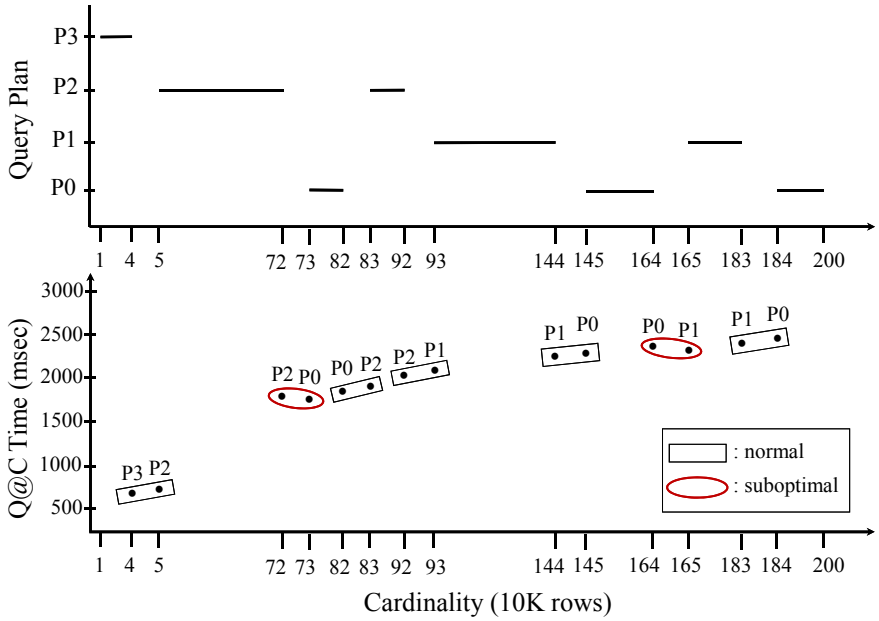


Fig. 1. An Example of Suboptimality and Fluttering

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.

In Section 3 we briefly summarize the vast amount of related work in query optimization to establish the technical basis for our study. The following section introduces the methodology we will follow, that of *empirical generalization* [**?**]. We then present in Sections 4 and 5 a predictive, causal model of suboptimality and state seven specific hypotheses derived from that model. We then test these hypotheses across a number of queries, data, and cardinalities, whose results provide strong support for the validity of the proposed model. These are the first scientific results that we are aware of that apply *across* DBMSes, rather than on a single, specific DBMS or on a specific algorithm. In Section 7 presents a follow-on analysis that demonstrates a fundamental limit to the number of operators that a cost-based optimizer can support, and indicates that perhaps extant DBMSes have hit that limit. This model has implications listed in Section 8 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.id4, 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 plan 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 P1 at 1,650,000 to P0 at 1,640,000 tuples. Such pairs identify suboptimal plans. For the pair at 720,000 tuples, P0 required 2.35sec whereas P1 at a larger cardinality required only 2.41sec. This query exhibits seven plan change points, two of which are suboptimal.

This query also illustrates an interesting phenomenon, in which the optimizer returns to an *earlier* plan. Sometimes 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 6.3, we started with 6,967 query instances (a query run on a specific DBMS) after an extensive query measurement protocol [5; 6] applied its extensive sanity checks. While about 20%, or 1,491, of these query instances contained only one plan, a few of the other query instances switched plans at almost every change in cardinality (we varied the cardinality in increments of 10K tuples, a total of 200 cardinalities): see Figure 2. Slightly over half, or 3,933 query instances, 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.

Fig. 2. Cumulative percentage of queries exhibiting the indicated number of plan changes

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. (Thus, the very term "query optimizer" is aspirational rather than accurate.) 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 2,738 query instances, or 67% of the suboptimal queries, exhibited only a small degree of suboptimality: less than 30%. The challenge is that over fifth of all queries (1,355) exhibited a significant amount of suboptimality ($\geq 30\%$). The relative slowdown can thus be quite large for some queries.

We started with 7,640 query instances, each of which is a particular query running on a particular DBMS, cf. Experiment 7 of Table I in Section 6.3. After our protocol, we were left with 6,967 query instances, of which 5,475 had at least one change point, so those are the ones we consider further. Of those, 1,382 had *no* suboptimality, so 4,093 (75%) had some suboptimality.

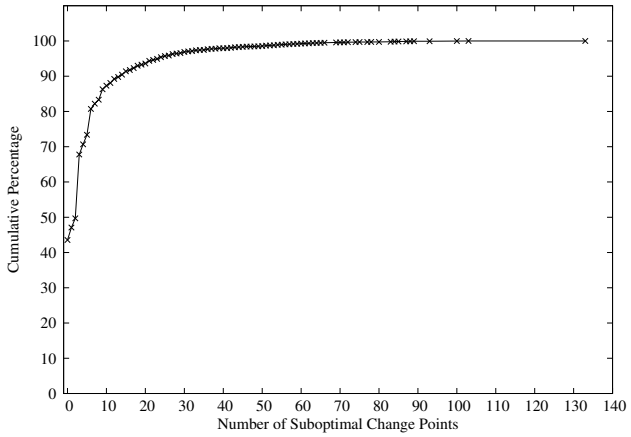


Fig. 3. Cumulative percentage of queries exhibiting the indicated number of changes to a *suboptimal* plan

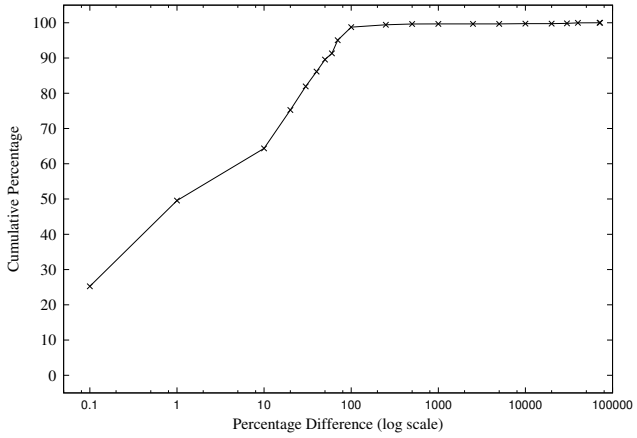


Fig. 4. Cumulative percentage of queries exhibiting the indicated percentage relative suboptimality

1,951 (36% of the query instances with a change point) have the higher cardinality running at least 20% faster than the lower cardinality. That means that 2,142 query instances (slightly over half of the suboptimal queries) were barely suboptimal (<20% slower) and about a third (1,355) were considerably suboptimal (where the higher cardinality ran at least 30% faster than the lower cardinality).

We emphasize four important points.

— First, we used a sophisticated query measurement methodology that reduces the measurement variance, 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. We view query optimization in modern DBMSes as 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.
— Fourth, we wanted to understand whether a fundamental limitation exists in the prevalent approach to query optimization utilized by DBMSes generally, and certainly by the four DBMSes that we studies.

This is why we needed a new methodology. We want to understand cost-based optimization deeply. This means that we need to go beyond the examination of a single query optimizer, as is done in almost every paper over the forty-year history of query optimization research, to study multiple instances of that optimization architecture, in an effort to achieve generalizable results.

Thus, in this paper, we articulate a predictive causal model for how and in what circumstances suboptimality arises and provide compelling evidence via hypothesis testing that this model accurately characterizes the behavior of query optimizers in general. This is what is meant by "empirical generalization" [**?**] and why it is needed to answer such questions. In the long history of research in database query optimization, or even of databases in general, our model and its hypothesis tests are the first predictive results that we are aware of that apply *across* DBMSes, rather than on a single, specific DBMS or on a specific algorithm. Such a DBMS-agnostic, though paradigmatic, 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

TODO: *Rick: For Science article, add history: IMS-¿System R, procedural-¿declarative, Selinger's query opt a*

TODO: *Rick: For TODS, shorten history, probably remove SQL cites, just one sentence about central role of Se*

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 [20] and an even larger extension as SQL:1999 [18] and refinement as SQL:2008 [14] and SQL:2011 [19].

There has been extensive work in query optimization over the last 40 years [12; 15], during which a particular quite effective paradigm had taken hold, in both open-source and proprietary DBMSes. In this over-arching paradigm, query optimization and evaluation proceeds in several general steps [22]. 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* [24]. 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 [7].

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 [13; 17], 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. This objective must be ensured both by the initial SQL-to-relational algebra translator and by the subsequent query enumerator. A secondary but clearly very important objective is *efficiency*; after all, that is the raison d'etre for this phase. As is well known and has been alluded to already, 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; 16], 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" [25, page 59]. And as we will see, suboptimality may occur in sophisticated query optimizers even when considering only simple queries.

While this paper does not provide direct solutions to address suboptimality, we envision that by following up on the implications of our proposed predictive model for suboptimality, engineering practice, such as dynamic reoptimization just mentioned, may benefit. We elaborate on this subject in Sections 6.8 and 8, 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 or hundreds of thousands 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 [14], 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 [16] 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

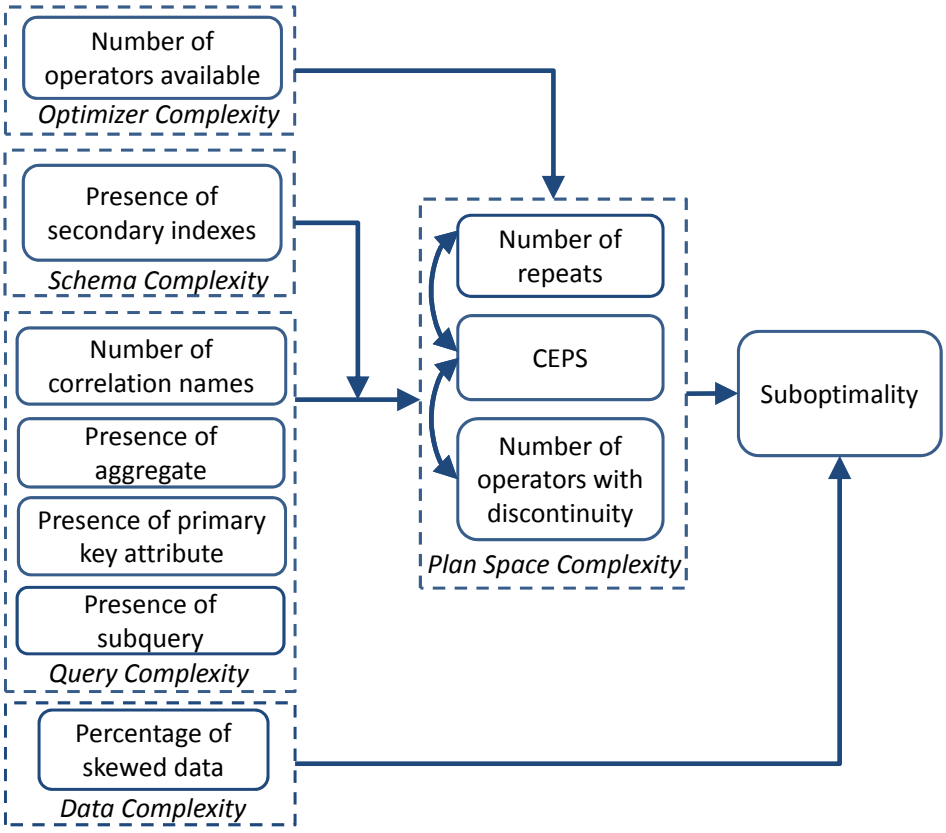


Fig. 5. Predictive Model of Suboptimality

object-relational systems [18]. 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 tens of thousand 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 variable will have on the one dependent variable, query suboptimality (with some of the the influences mediated by one of the constructs). In the next section we will operationalize these variables, explaining how each is controlled or measured.

4.1. Constructs in the Model

The model concerns five general constructs that we hypothesize will play a role in suboptimality: optimizer complexity, schema complexity, query complexity, data complexity, and plan space complexity. Two 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.

Our theory, which we will investigate in some detail and which is the basis for our predictive model, is that suboptimality is due in part to the complexity of the optimizer and concomitant unanticipated interactions between the different components used within the optimizer with various sources of complexity in producing a good (hopefully optimal) plan. We argue that with the proliferation of concerns and variability that an optimizer must contend with, it is extremely difficult to ensure that for an arbitrary query that the optimizer will *always* pick the right plan. There are many sources of complexity that may challenge one or more components of the optimizer; our theory implicates specific sources as contributing to observed suboptimal plans.

The model depicted in Figure 5 suggests specific factors that 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 suboptimal at any cardinality of the input table. In Section 5, we delve into the details of how we operationalize this and the other variables we now examine.

4.2. Variables in the Model

The model has several independent variables which influence suboptimality.

For the construct of *optimizer complexity* we have one independent variable, "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 with in query plans.

We have one variable for the construct of *schema complexity*: "Presence of secondary indexes." If true, that means that every non-primary-key attribute, for the table for which we vary the cardinality (termed the *variable table*) as well as for the other three tables (termed the *fixed tables*), has a secondary index associated with it. The rationale is that the presence of secondary indexes expands the number of possible plans: each predicate in the query can often be mapped to one or more operators utilizing that index.

For the construct of *query complexity* we have identified four variables: "Number of correlation names", "Presence of aggregate", "Presence of primary key attribute", and "Presence of subquery". For each independent variable, we have interventional control in our experiments, in that we can manipulate the values of these variables through the construction of the actual query to be optimized. The first is the number of correlation names defined in the FROM clauses. The second is whether a single aggregate operator (that is, either `Max` or `Avg`) appears in the SELECT clause. The third is whether a primary key attribute appears in at least one predicate in the WHERE clause. The fourth is whether a single subquery appears in the WHERE clause; that subquery evaluates to a single value that is equality-compared with an attribute, in the where clause. The rationale is that each factor may expand the number of plans or otherwise render the search for the optimal plan more complex.

We have one variable for the construct of *data complexity*: that of "Percentage of skewed data". Skew is generally defined as how values are distributed with a column of a table. We refine this to "how many duplicate values are present," with zero skew meaning that there are no duplicate values present and 100% skew implies that there is but one value in the entire column. The presence of skew complicates query time estimation, which in turn complicates the search for the optimal plan.

In the middle of the figure is the construct of *plan space complexity*. Given a particular query, its complexity will impact the total number of candidate plans considered by

the optimizer. This variable 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 the "effective plan space" and term 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" [11], though we focus on a more specific aspect of discontinuity.) Finally, the variable "Number of repeats" *is* directly measurable as the number of times a plan is reused across the cardinality of the variable table. For example, if the sequence of plans for a query as the cardinality increases is $A\ B\ B\ B\ C\ C\ A\ B\ B\ B\ C\ B\ C\ C\ C\ B$, then the "Number of repeats" will be six: removing sequential duplicates results in $A\ B\ C\ A\ B\ C\ B\ C\ B$, with the last six distinct plans being duplicates of the first three. Interestingly, CEPS plus the number of repeats gives you the number of plans in the sequence, again, after removing sequential duplicates, nine plans in this particular case. Thus, we associate with the latent construct of plan space complexity three measurable (that is, indirect) variables. Our model stipulates that the construct, and thus the three associated with this construct, intervene between the constructs of optimizer complexity, schema complexity and query complexity and the construct of suboptimality.

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

4.3. Interactions in the Model

Given these six constructs and eleven specific variables depicted in Figure 5, let's now examine the causal relationships between these variables, depicted as directed arrow between variables (a line originating or ending at a construct is interpreted as originating or ending at all variables in that construct). Such relationships are specific *interactions* between the the constructs (in particular, between 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 indirectly, via plan space complexity. We hypothesize that an optimizer with a larger number of available operators will generate more plans and hence increase plan space complexity.

Hypothesis 1: Number of operators available will be positively correlated with, (a) Number of repeats, (b) CEPS, and (c) Number of operators with discontinuity.

We now turn to query complexity, a construct associated with four independent variables. As with the optimizer complexity construct, we include in our model an indirect interaction through plan space complexity.

A higher value of each of these specific variables implies a more complex query. We expect a strong relationship between Number of correlation names to CEPS (because it is well-known that the number of potential join combinations is exponential to number

of correlation names), to number of repeats (number of repeats could partially track CEPS), and number of operators with discontinuity (for the same reason).

We also expect a positive relationship between presence of aggregate (as that will definitely add at least one operator to the query plan), presence of primary key attribute, and presence of subquery.

Hypothesis 2: Number of correlation names will be strongly correlated with (a) Number of repeats, (b) CEPS, and (c) Number of operators with discontinuity. Presence of aggregate will be positively correlated those those three variables (correlations d–f) and similarly with Presence of primary key attribute (correlation g–i) and Presence of subquery (correlations j–l).

We don't feel that skewed data (in the "data complexity" construct) will impact plan space complexity, but rather that it will negatively impact accuracy of plan cost estimation, and thus increase suboptimality.

Hypothesis 3: Percentage of skewed data will be negatively correlated with Suboptimality.

Let's now turn to the plan space complexity construct. 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.

The model also has two interactions within plan space complexity, between CEPS and number of operators with discontinuity variables and between number of repeats.

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 suboptimality 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. Any inaccuracy in the estimation of plan statistics may lead the optimizer to select a suboptimal plan.

Hypothesis 4: Plan space complexity (CEPS) and (a) Number of operators with discontinuity and (b) Number of repeats will be positively correlated.

Similarly, a plan chosen by the optimizer once will be reconsidered at a later cardinality and perhaps chosen, following the past experience of selecting and using that plan among many other plans. Even if many different plans are already used, for the same reason the optimizer may revisit a pool of the previously used plans and choose one of them than to explore other new plans, given the complexity of plan cost estimation. Then the larger CEPS, the more times the optimizer repeats using the same plans. We thus hypothesize that the number of repeats has a positive correlation with CEPS.

Hypothesis 5: The Number of repeats will positively correlate with CEPS.

The schema complexity construct consists of the variable "Presence of secondary indexes." Such indexes provide opportunities for the optimizer to consider more candidate plans that use these indexes, due to the additional query evaluation operators now applicable. Those additional plans enable 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 secondary indexes

is to increase the strength of the interaction between query complexity to plan space complexity.

Hypothesis 6: Presence of secondary indexes will strengthen the correlations between the Query Complexity construct, specifically, the variables Number of correlation names, Presence of aggregate, Presence of primary key attribute, and Presence of subquery, and the Plan Space Complexity construct, specifically, variables Number of repeats, CEPS, and Number of operators with discontinuity (correlations a–l listed above in Hypothesis 2).

Finally, we hypothesize that queries with a high plan space complexity present challenges to the query optimizer, and thus increase the chance that the query optimizer picks a suboptimal plan. This is a fairly direct connection for CEPS and Number of operators with discontinuity. For the Number of repeats variable, a query exhibiting a high number of repeats has more opportunities for suboptimality at some cardinality, just because the number of repeats is an indication that the query optimizer is struggling.

Hypothesis 7: Suboptimality will be positively correlated with Plan space complexity, that is, (a) Number of repeats, (b) CEPS, and (c) Number of operators with discontinuity.

We have just *described* how suboptimality might arise, through a theory and its elaborated causal model, which implies seven specific hypotheses. 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 independent variable is a property of a DBMS (number of operators available), of the schema (presence of secondary indexes), of a query (number of correlation names, presence of an aggregate, presence of primary key attribute, and presence of subquery), of the data (percentage of skewed data), and of the plan space (number of repeats, CEPS, and number of discontinuous operators). There is one dependent variable of our model (suboptimality).

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, thereby moving up the y-axis of Figure 6.

5.1. Optimizer Complexity

By “number of operators available 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

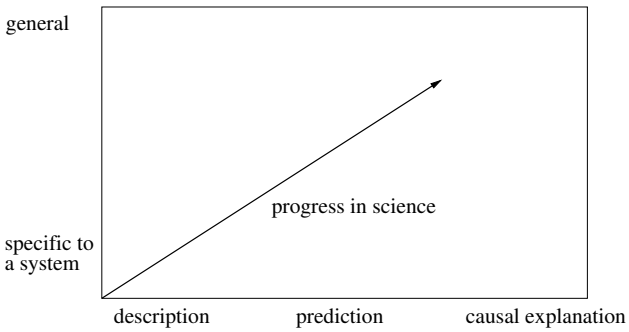


Fig. 6. Empirical Generalization

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 8 to 53 across all the queries and data sets used in the Exhaustive, Exhaustive with Keys, and Exploratory Experiments, to be discussed in Section 6.3.

5.2. Schema Complexity

The presence of secondary indexes is easy to operationalize. We generate two databases, one without any secondary indexes and one with a key specified for each table on the non-key (that is, other than the first) attribute.

5.3. Query Complexity

Query complexity is also relatively 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 one aggregate functions in the SELECT clause. As such, some of the complexities mentioned by Kabra and DeWitt [16], such as user-defined data types, methods, and operators, are not considered. Concerning presence of primary key attribute, if this independent variable was set to 1, we ensured that there was at least one primary key attribute in one of the comparisons in the WHERE condition.

The queries were generated by a simple algorithm. The SELECT clause will contain from one to four attributes, hence, an average of 2.5 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, the presence of an aggregate is false (0), the presence of primary key attribute is false (0), and the presence of subqueries is false (0).

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 ran-

dom 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. This is realized by setting the attributes `maxIsAbsolute` to `true` and `complexUsePercentage` to `100`. Basically, "complex" predicates eliminate the Cartesian product, and by setting complex predicates as "absolute," no additional predicates are included except for those which are necessary for eliminating Cartesian product. Also for simplicity, we include neither disjunctions nor negations.

The presence of subquery was 0 or 1, with 0 indicating no subquery. For each query needing a subquery, we picked a separate generated query and rendered it as a subquery to replace an attribute in the where clause of the original generated query. As an example when this variable was one, we started with query qs1-2, shown here.

```
SELECT t0.id2, SUM(t1.id3)
FROM ft_HT2 t0, ft_HT2 t1
WHERE (t0.id2=t1.id1)
GROUP BY t0.id2
```

We then generated another simple select-project (SP) query at random concerning the variable table and simply replaced one side (in this case, the right side) with the generated entire query as a subquery, to produce this final query.

```
SELECT t0.id2, SUM(t1.id3)
FROM ft_HT2 t0, ft_HT2 t1
WHERE (t0.id2 IN (SELECT t2.id3 FROM ft_HT1 t2))
GROUP BY t0.id2
```

We thus have a maximum of only one level of nesting.

5.4. Data Complexity

This independent construct has one variable: percentage of skewed data.

Skew has a very specific definition in statistics, involving elongating the left or right tail of a distribution, thereby moving the mean left or right of the median (in a symmetric distribution the mean = median = mode).

But we start with a distribution without a tail: the uniform distribution: the values from 1 to 2 million (2M). We consider this to be a skew of 0 (no skew). At the other end of the spectrum is one in which all the values are identical, or a skew of 1.0.

We define the skew quite simply: "the reciprocal of the number of distinct values," so $0 < \mathit{skew} \leq 1$. For 2M distinct values, the skew would be $\frac{1}{2M} = 0.000005$, which is practically 0. For exactly one distinct value, the skew would be 1.0. For two distinct values, the skew would be 0.5. For ten distinct values, the skew would be 0.1.

We can generate the table of 2M rows by generating values sequentially from 1 to the number of distinct values. This creates a "span of values." We repeat this for the second span if necessary, and on and on, until we have 2M values in all.

When varying the cardinality, we remove 10K values from the variable table and then copy those tuples to a new table to ensure that every page is as full as possible (that is, 100% load factor). This gives us a table of 1.99K tuples. (We then get a query plan for this table.) We repeat this removal process until the final cardinality reaches 10K.

The way we effect the 10K removal is as follows. The key idea is to remove individual spans until we've deleted 10K values. Since we don't touch the remaining spans, the number of distinct values does not change, and so the skew remains constant.

Let's first illustrate with a skew factor of 1.0. This translates to exactly 1 value for the entire table, or 2M spans. To shorten, we remove 10K spans, equal to 10K values.

This removal keeps the skew at 1.0, due to the unique values in the remaining spans. At the end, 10K spans, each with a single value, remain.

What about a skew factor of 0.1? This skew factor translates to 10 distinct values. So we generate 200K spans, each with 10 values. To shorten, we delete 1,000 spans, which is equivalent to removing 10K values. As before, this does not change the skew. At the end, the table will have 1,000 spans with 10K values, retaining a skew factor of 0.1.

Let's see an example at the other end of the spectrum, of a skew factor of 1/2M. This skew factor, which is almost close to 0, gets translated to 2M distinct values. So we generate a single span of 2M values. To shorten the table, it makes no sense to remove this single span in its entirety. But we can remove 10K values from this single span. Note that this changes the skew to 1/1.99M, then eventually to 1/10K, which is still very close to zero (the skew has changed from .0000005 to .00001).

We use five skewness values: 0.000005 (which we term only slightly misleadingly as "no skew"), 0.001, 0.1, 0.5, and 1.0, for this independent variable.

5.5. Plan Space Complexity

This explanatory construct includes three 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 4.

The number of repeats is just the number of times a plan associated with a smaller cardinality is repeated, after removing sequential duplicates.

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 85 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. 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.

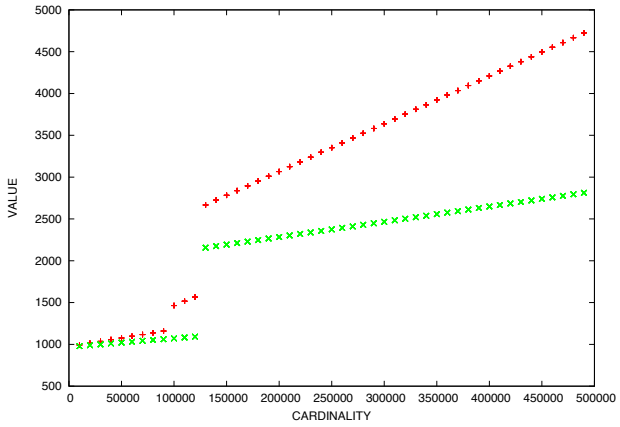


Fig. 7. An Example of Discontinuous Plan Operators (Hash-Join)

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 adjacent 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 adjacent 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 7 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 (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 adjacent 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 7, 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.

5.6. 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 [8; 9].) 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.

We modify the cardinality to produce multiple execution plans for a given query. For one of the DBMSes, we can modify the stored table statistics directly. For the other DBMSes, we had to do so indirectly, by varying the size of the table and running the optimizer on tables of different size. As we vary the cardinality, we collects all the plans that the optimizer felt were appropriate for that query.

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

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 6.3). We varied the cardinality of the variable table by starting with the maximum size, running the queries, then deleting 10K tuples and repeating. We performed an "`ANALYZE TABLE`" function to force the DBMS to update the table's statistics to be accurate before actually evaluating the query (we did this for all experiments). We expected that as the cardinality decreased, the runtime would also monotonically decrease. However, due to the variance in query time measurement observed even when the cardinality was identical, we encountered spurious violations.

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 relaxed our definition of monotonicity to the following.

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 $\forall p \in \textit{plans}(p)\ \forall c' > C$,

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

As we will see in Section 6.3, we observed only 3,347 violations (0.73%) of non-strict monotonicity, for the largest experiment (Experiment 7: Confirmatory) across all the DBMSes we studied, justifying our conclusion that the DBMSes under study are indeed 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 (designated as $n-1$) and at the change point (that is, n).

The query is said to be suboptimal if $\textit{time}_{n-1} - 0.5 \cdot \textit{stddev}_{n-1} \geq \textit{time}_n + 0.5 \cdot \textit{stddev}_n$. At each Q@C pair, Suboptimality is coded as four levels (0–3), based on the distance in standard deviations, up to three standard deviations (we chose this cutoff because 99.7% of all Q@C pairs are less than or equal to three standard deviations). We then sum this value over all Q@C pairs with different plans (the change points) to arrive at a single integer for the query instance.

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 6.3, the value of suboptimality ranged from 0 (no suboptimality) to 133, with the majority between 0 and 9. A full 56% of the queries were suboptimal. Because the occurrence of large values of this measure was so rare, we did a log transformation: $\log_{10}(1 + \textit{subopt})$ in the confirmatory analysis.

6. TESTING THE CAUSAL MODEL

6.1. Experimental Setup

The measurements were collected using Tucson Timing Protocol Version 1 (TTPv1) [5] and Version 2 (TTPv2) [6] on a suite of five machines, each an Intel Core i7-870 Lynnfield 2.93GHz quad-core processor on a LGA 1156 95W motherboard with 4GB of DDR3 1333 dual-channel memory and Western Digital Caviar Black 1TB 7200rpm SATA hard drive, running Red Hat Enterprise Linux Server release 5.8 (Tikanga) for TTPv1 and release 6.4 (Santiago) for TTPv2, with

a kernel of 2.6.32-358.18.1. The protocol provided calculated query evaluation time, including computation and I/O time, in msec. Both protocols were utilized exactly as specified.

No run violated the experiment-wide sanity checks. For the largest experiment, the Confirmatory experiment 7 discussed below, approximately 10.5% of the query executions and 5.1% of the queries-at-cardinality (Q@Cs) were dropped due to query execution and Q@C sanity checks. As a result, excessive variation in calculated query time was observed in only 0.003% of the Q@Cs. A total of 0.35% of Q@C adjacent pairs violated relaxed monotonicity and 0.68% of the Q@Cs violated strict monotonicity, which is acceptable.

We thank the developers of this protocol and of the software that enabled the running of many, many queries for providing us this software.

In these experiments, we installed each disk directly on the machine that also runs the DBMS, ensured a cold cache (disk drive, disk controller, O/S, and DBMS buffer), and discarded any sequence of query executions that appear to be the result of query result caching. We also ensured within-run plan repeatability.

In the following, we describe in detail the data used by our experiments and the experimental scenarios we defined. More details on the data sets and experiment scenarios used can be found in Appendix A.

6.2. Data sets

We generate our experiment data set randomly in each of the experiments. However, we use seeds to control the random data generator so that it can produce repeatable data as required.

Our experiment data set 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, physically, of such tables as the experiments are being performed. We organized three configurations for the data sets. The first configuration is a small data set with four tables, each with four integer-typed attributes. Each table is populated with one million rows, and thus the size of each table is roughly 16Mbyte. We also produced a version of the data set with primary keys (of the first attribute) for all tables.

6.3. 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.

Utilizing JDBC to manipulate independent variables from outside the DBMS allows us to empirically generalize by moving up the y axis of Figure 6, from one system to several systems and then to a general theory. We don't reveal the identity of the DBMS we studied, for two reasons. First, commercial DBMSes include in their user agreements requirements not to release performance data. This is detrimental to science, but we have no choice but to live with that restriction. However, in some sense the specific DBMS doesn't matter, as we are studying phenomena about cost-based optimizers *in general*, and so are interested in making statements that apply to all the experimental subjects in our study.

Table I. Experiments 1–7: Detailed Run Statistics

	Experiment	*Protocol*	*Cumulative Hours*	*Number of Query Instances*	*Number of Q@Cs*	*Number of QEs*	*Number of Retained QEs*
1	Monotonicity	—	38	60	12,000	12,000	12,000
2	Initial Exhaustive	TTPv1	1,672	160	32,000	320,000	244,787
3	Exhaustive	TTPv2	1,544	160	32,000	320,000	319,980
4	Exhaustive with Keys	—	28	200	40,000	—	—
5	Initial Exploratory	TTPv1	560	780	8,842	88,420	68,891
6	Exploratory	TTPv2	1,663	1,200	12,560	125,600	114,377
7	Confirmatory	TTPv2	11,375	7,640	99,558	995,580	890,631
	Total		16,880	10,200	236,960	1,861,600	1,650,666

Table I exhibits the statistics of running a number of queries in our experiments. All but the last column list the number of query instances, etc., gathered by the protocol for each experiment. The last column gives the number of query executions retained by the protocol. One of the primary goals of TTPv2 was to reduce the number of QEs discarded due to phantom processes that were indirectly detected. Experiments 3, 6, and 7 benefited from that protocol.

We performed six 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 5.6. 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.

When TTPv1 became available, we performed our second experiment, termed "Initial Exhaustive," which more accurately tested the monotonicity assumption, as also described in Section 5.6. 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 Initial 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 later reran this experiment using TTPv2 (we term this simply "Exhaustive"). Note that the number of retained QEs went up a lot (that is the main benefit of TTPv2), while the number of hours required actually went down (another benefit).

We also ran an experiment (number 3 in the table) 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 initial exploratory analysis of a prior version of the causal model. (This fourth experiment used the first version of the Tucson Timing Protocol (TTPv1) [5].) The earlier model had fewer independent variables yet also had more complex interactions. Specifically, the model did not include the independent query complexity variables of presence of secondary indexes, presence of subquery, nor the schema complexity independent variable of presence of secondary indexes, nor

the plan space complexity independent variable of number of repeats. The prior model also had presence of primary keys in the schema complexity construct rather than the query complexity construct. In reformulating that independent variable, we were able to remove a complex mediating moderator between the plan space complexity and suboptimality constructs.

The fifth experiment was an "Initial 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 data set, for exploration across all combinations, a total of 780 query instances. These query sets are enumerated in more detail in Appendix A. We ran the DBMSes on the *change points*: adjacent cardinalities (10K apart; 300 for one DBMS) with different plans. The protocol retained about 78% of the QEs.

We later reran the exploratory analysis using TTPv2, this time on a larger sample of queries and data sets: (a) 200 queries of four DBMSes without primary keys defined and (b) 100 queries from each of the four DBMSes on the primary key data set, for exploration across all combinations, a total of 1,200 query instances (= 300 $\times$ 4 (DBMSes)). We termed this sixth experiment "Exploratory." We ran the DBMSes on the *change points*: adjacent cardinalities (10K apart; 300 for one DBMS) with different plans. This experiment required 1,663 hours. The experimental methodology retaining 12,100 Q@Cs (3.7% were dropped), covering 1,123 query instances (6.4% were dropped). Only 443 strict monotonicity violations (0.78%) and 301 relaxed ones (0.54%) were observed. This exploratory analysis allowed us to refine the operationalizations.

The seventh 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) 510 of those queries that had joins on the primary key attributes, on the data set with primary keys, (c) 100 queries on the data set without skew and primary keys, (d) 100 queries with a subquery on the data set without primary keys, (e) 100 queries on the data set with primary keys and secondary indexes defined, (f) 100 queries with a subquery on the data set with primary keys defined, (e) 100 queries with a subquery on the data set with primary keys and secondary indexes defined, and (g) 100 queries on the data set with primary keys and secondary indexes defined, for a total of 1,910 queries and a total of 7,640 query instances (over the four DBMSes). Again, we ran the DBMSes at the 99,558 Q@Cs that were observed, roughly 13 per query. The protocol accurately timed both sides of adjacent Q@Cs (53,547 change points in all).

In this Confirmatory Experiment, we observed 3,347 strict monotonicity violations (0.74%), and 1,966 relaxed monotonicity violations (0.43%) (out of a total of 452,684 pairs of two Q@Cs having identical plans and query instance), which provides further confidence that monotonicity also applies to operators found in queries over data in various contexts (as described above) and that our operationalization of suboptimality is a valid one.

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

6.4. Descriptive Statistics

Several initial conclusions can be drawn from this confirmatory experiment, which was the result of several years of programming effort and about 30 months of experimental runs, summarized in Table I.

Let illustrate the descriptive statistics with the Confirmatory experiment. In that experiment, we started with 7,640 query instances. These resulted in 99,558 Q@Cs,

with 10 each query executions (QEs). There were 890,631 retained QEs, which resulted in 6,983 retained query instances (equivalently, 657 query instances were dropped by the protocol). Additionally, 16 query instances were dropped *after* the protocol because they were missing data for the only change point in the query. Therefore, we were left with 6,967 queries.

First, perhaps surprisingly, more than half (3,933 queries out of the 6,967 query instances that emerged from the measurement protocol) exhibited suboptimality somewhere in the range of cardinality of the varying table. *Every* DBMS exhibits suboptimality.

Secondly, most (3,370) of those suboptimal queries (86%) had the maximum value of suboptimality (i.e., level 3) at some Q@C pair.

This phenomenon (query suboptimality) 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.

Third, concerning the causal factors of suboptimality in our model,

— the cardinality of the effective plan space (CEPS) ranged from 1 to 24 plans across the cardinality range,
— the number of discontinuous operators observed in plans for a query averaged 9.4, with 85 being the maximum, and
— the number of repeats ranged from 0 to 108, with the mean being 4.99.

6.5. Correlational Analysis

We tested Hypotheses 1–7 using the strength and significance of correlations of variables involved. These hypotheses predicts eleven positive relationships. Table II 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.)

TODO: *Sabah: update after filling out Table II:* 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 available and suboptimality was negative, while we predicted positive correlation. Hypothesis 2 was not significant.

6.6. Regression Analysis

TODO: *Sabah: revisit this entire section, with confirmatory analysis numbers* We ran a regression over the independent variables of the model that predict suboptimality over the data from the Confirmatory experiment. Our model explained % of the variance of the suboptimality dependent variable.

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

Hypothesis 6 predicts that the presence of secondary indexes will be a moderator, strengthen the interactions between query complexity with plan space complexity. It thus provides predictions that the strength of such interactions will increase in the presence of secondary keys. When secondary indexes were not specified, our model explains % of the variance of CEPS, and % of the variance of discontinuity, with all of the independent variables. When secondary indexes were defined on the underlying tables, our model explains % of the variance of suboptimality, % of the variance of CEPS, and % of the variance of discontinuity. These findings are all consistent with our hypothesis predicting a positive moderation effect of primary keys.

Table II. Testing Hypotheses 1–7: Correlations on the Confirmatory Study

Variable	*Suboptimality*	*Repeats*	*CEPS*	*Discontinuity*
Operators in DBMS	—	**H1a:**−0.16	**H1b:**	**H1c:**
Correlation names		**H2a:** 0.54	**H2b:**	**H2c:** 0.49
Skew	**H3:**	—	—	—
Presence of aggregate		**H2d:**	**H2e:**	**H2f:**
Presence of primary key		**H2g:**	**H2h:**	**H2i:**
Presence of subquery		**H2j:**	**H2k:**	**H2l:**
Repeats	**H7a:**	—	**H5:**	—
CEPS	**H7b:** 0.	**H4b:**	—	**H4a:**
Discontinuity	**H7c:** 0.31	—	—	—

Table III. Testing Hypothesis 6: Interaction Strength on the Confirmatory Study

Variable	*Repeats*		*CEPS*		*Discontinuity*	
	Not SI	SI	Not SI	SI	Not SI	SI
Correlation names		0.	0.	0.	0.	0.
Presence of aggregate	0.	0.	0.	0.	0.	0.
Presence of primary key		0.	—	—	—	—
Presence of subquery		0.	—	—	—	—

Table III shows that the strength of all interactions goes up as predicted, except for ones involving 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.

6.7. 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 seven specific hypotheses listed in Section 4. We then performed six experiments,

— to refine our operationalizations: specifically discontinuity, via the Initial Exhaustive and Exhaustive Experiments, and number of operators available, via the Exhaustive with Keys Experiment,
— to test fundamental assumptions, specifically monotonicity, via the Monotonicity Experiment, and
— to test and make minor refinements to our model: the Initial Exploratory and Exploratory Experiments.

During this exploration, when the TTPv2 protocol became available, we reran experiments to avail ourselves of the increased precision due to a much higher percentage of retained QEs and thus Q@Cs and query instances.

Throughout these six experiments, we were 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 such errors, we then performed the final Confirmatory Experiment on a completely different data set consisting of many more query instances, 7,640 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.

TODO: *Sabah: update with final results:* 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 available remains largely 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.

Now that the causal model has been found to be supported by the confirmatory analysis, we turn to possible implications of this model.

6.8. 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.

TODO: *Sabah: update with final numbers:* 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 II: 0.51–0.54, normalized) The next highest regression factor was number of tables involved, which is highly correlated with CEPS (cf. Table II: 0.54). These two observations imply that the number of plans being considered is a major determinate of suboptimality, indicating that choosing among many plans is hard, despite many decades of research and development.

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

The one following that is the number of operators provided by the DBMS. This factor is totally correlated with the DBMS, and so is confounded with other specificities of the coding of each DBMS. That said, it is still the case that as the number of operators goes up, across all the DBMSes studied, the amount of suboptimality increases. This factor also strongly impacts CEPS, and so indirectly impacts suboptimality through that path.

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

7. DIMINISHING RETURNS?

In order to increase query performance, DBMSes are extended over time with new relational query operators. Also over time, DBMSes are extended with new storage and indexing structures, which themselves elicit new query operators. For example, nested loop join was probably the first implementation of this needed algebraic operator. B-tree index join was only possible when such indexes were added (quite early in the history of modern DBMSes). Adding hashing to a DBMS enables a hash join operator; adding hash indexes enables (perhaps several variants of) hash-index join. Each subsequent generation of the DBMS thus supports an ever-expanding collection of operators, with the current incarnation the most recent within a series of DBMS *generations*.

7.1. A Gedanken Experiment

Consider a Gedanken experiment that examines the plan for each query at each cardinality, that is, each Q@C, *for each DBMS generation*. In early generations, there will be few plan changes for a given query as the cardinality varies simply because there are few operators available. For subsequent generations, some of the Q@Cs will be associated with different plans, enabled by the new operators that were added.

In many (hopefully most) cases, a new plan selected by a generation will be more efficient than the plan selected by the immediate previous generation. After all, that is the very reason the new operator(s) were added to that subsequent generation. How-

ever, in the presence of suboptimality, sometimes the new plan at that Q@C is *not* preferable, as that DBMS generation's query optimizer selected the wrong plan. Indeed, the query optimizer also evolves and improves with each new generation, in part to minimize the chance of selecting the wrong plan.

Our predictive model in Figure 5 suggests that suboptimality is causally impacted by the independent variable of CEPS (cardinality of effective plan space), and CEPS is naturally connected to the number of operators available **TODO:** *Rick: connect with Section 6.8 after Sabah updates that section; also discuss the exponential incre*

A possible implication is that as the evolution of generations of the DBMS add more and more operators, suboptimality will also increase.

It is important to emphasize that our causal analysis to this point has been *across multiple DBMSes*, each with a different set of operators and even number of operators. Let's reprise from Section 1 on page 1: Our goal, up to this point, has been "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."

The present discussion has a quite different focus. Here we're talking about the query optimizer *within* a single DBMS, across multiple generations, each generation adding one or more operators.

Let's then refine our Gedanken experiment. Let's assume each subsequent generation adds a single operator. Starting with a fixed set of queries, for each DBMS, we run each generation on each Q@C and then sum up the query times to compute a *per-generation (total) time* for that DBMS. We can also sum over all four DBMSes, to see how this *overall per-generation time* (a single number for each generation number) varies with generation.

The underlying question then becomes, does an additional operator made available in a subsequent generation of the DBMS actually help or hurt? More fully, is the predicted increase in suboptimality originating from that added operator, a causal effect discovered and validated by the process of *science*, specifically the application of empirical generalization illustrated in Figure 6, compensated for by the increased performance afforded by that operator, a benefit realized by the application of *engineering*, the articulation, elaboration, and implementation of new storage structures and query evaluation algorithms?

Fortunately, the engineering has already been done: current DBMSes have at their disposal multiple query operators, which have been perfected over the decades. Adopting the engineering perspective would predict that the overall per-generation time would *monotonically decrease as the generations add query evaluation operators*.

Interestingly, adopting the scientific perspective reaches a different conclusion. Our causal model in Figure 5 predicts that as the number of operators increases, the latent measures of plan space complexity increase, which therefore also *increase* suboptimality. As the generations of the DBMS contain successively greater numbers of operators, suboptimality will also increase with generation number. Given that each new operator will improve a *shrinking* subset of Q@Cs for any given query, while the suboptimality occurs in a portion over an *expanding* subset of Q@Cs, our causal model implies that the curve of per-generation time will fall with each successive generation, but then start to level off as suboptimality becomes more prevalent (specifically, the prediction is that the second derivative is either. Eventually, the per-generation time will either asymptotically approach a horizontal line (with the first derivative approaching 0 from below), or worse: the first derivative changing to positive, with the per-generation time over the queries we are studying actually *increasing* with DBMS generation. In either

case, the causal model predicts a point where *the increase in execution time due to suboptimality obviates the decrease enabled by the new operator*.

Put in simpler terms, engineering considerations alone imply that DBMSes can continue to improve, whereas also adopting the scientific perspective predicts that eventually each DBMS will hit a wall, beyond which improvement is not possible.

Does such a limit exist, and if so, how close are modern DBMSes to that limit?

7.2. Simulating DBMS Generations

While we do not have access to prior generations of our DBMSes (which is why the previous discussion was in the form of a Gedanken experiment), we can approximate this with an experiment using data already collected on the current version available for each DBMS, to *simulate* the prior generations, each successively having a fewer number of operators, and thus a smaller set of realizable query plans. (The effect will probably be smaller in our simulation, as we are nonetheless using the most recent query optimizer in all of the simulated generations; only the available operators will vary.)

In the next section, we'll explain how we characterize the generations of each DBMS. For now assume that each DBMS is associated with a series of generations, with each generation having one more operator. So generation 1 of the DBMS has just one operator, generation 2 has two operators, etc.

Our data consists of the 8,840 query instances and their 112,118 Q@Cs in both the exploratory and confirmatory experiments (Experiments 6 and 7) of the previous study. Specifically, we focus on consists of pairs of adjacent Q@Cs (let's refer to them as the *lower* Q@C and the *upper* Q@C, and thus we also have a *lower plan* and an *upper plan*) for the same query at the *lower* and *upper* cardinalities, that are adjacent (that is, separated by the minimum cardinality, either 10,000 or 300 rows), each with an actual execution time. All Q@Cs for that query and for that DBMS having cardinalities between the upper Q@C of one pair and the lower Q@C of the next pair are associated with that same plan. This will be the upper plan, guaranteed by the process in which we chose those Q@Cs for actually timing (recall from Section 2 that we start from the highest cardinality, that of 2M tuples, looking for changes in the plan).

For this generational experiment, we associate with each Q@C a generation (a positive integer) that is the earliest generation of the DBMS that contains all the operators in the plan. We also only consider Q@C pairs (with adjacent Q@Cs) where the lower plan has a generation distinct from that of the upper plan. Say the upper generation is earlier than that of the lower plan, as illustrated in Figure 8. In this figure, as we scan from right to left in decreasing cardinality, we first encounter Plan A from generation 2 at the highest cardinality of 1320 rows (this is the measured Q@C that has Plan A that is closest to the measurement at 910K rows), then a pair of Q@Cs with Plan A at 910K rows and Plan B at the adjacent 900K rows, then later Plan A again at 10K rows.

In this case, the thinking goes that, had we been in the DBMS generation 2, the optimizer would have selected Plan A throughout, because Plan B simply wasn't possible (as it involves an operator not present in generation 2). Then when generation 3 was created by adding an operator, the optimized chose Plan B for 900K. The reason Plan B was chosen is that it is faster than Plan A at that cardinality, shown in the figure by extrapolating the time down from 910K down to 900K (we will revisit this extrapolation shortly).

In this particular case, as just mentioned, Plan B is more appropriate (faster) than Plan A, but that is not the only possibility. Sometimes the later generation with more operators available chooses the *wrong* plan, one that is slower than the plan chosen

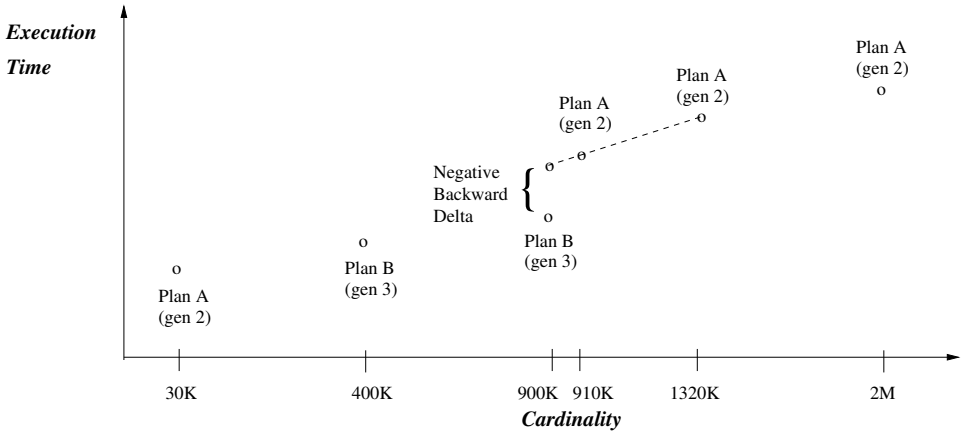


Fig. 8. An Example of a Negative Backward Delta

by the earlier generation, due to the suboptimality we've observed. (We'll examine an example shortly.)

The question then becomes, does a plan change by a subsequent generation (enabled by the additional operator(s) in that generation) represent a win (runs faster) or a loss (runs slower, because of suboptimality)? More broadly, do the Q@Cs in the aggregate enabled by each succeeding generation continue to overcome the increasing burden of suboptimality?

7.3. Characterizing DBMS Generations

To which generation do we assign each DBMS operator?

Note that we don't actually know the specific order in which the operators were added to each DBMS. But even if we did, that order was somewhat arbitrary, with a host of considerations going into those decisions over the years. Given that we are using the same optimizer for each such defined generation, we'll adopt a more systematic ordering of the operators. We start by gathering all *single-operator* plans, all *two-operator* plans, and so forth, and order the generation by the prevalence of their appearance in these query plans.

Specifically, for each DBMS we designate the first generation to contain the single operator that maximizes the number of plans at change points, that is, maximizing the number of Q@Cs, containing just that operator. So for example, all the plans generated by MySQL that have exactly one operator involve just the Full Table Scan operator. So no choice was needed: we designate the first generation as just having that one operator. Any plan with just that operator can be constructed by MySQL generation 1, as well as by any subsequent generation (as each generation includes that initial operator).

We then examine the plans containing exactly two operators. Using MySQL again, there are two such : one with the Full Table Scan and Full Table Scan with Join operators (888 Q@Cs) and one with the Full Table Scan and Ref operators (112 Q@Cs). Full Table Scan with Join is thus the operator added by Generation 2, given its prevalence of Q@Cs. Each subsequent generation adds that operator that maximizes the number of plans that operator will eventual enable. So Generation 3 adds the Eq_Ref operator, as that operator enables 633 plans eventually. Generation 4 adds the Ref operator and Generation 5 adds the Index operator. involve all five operators.

The generations of MySQL thus can be characterized from the Q@Cs we encountered: five distinct combinations of two operators, covering 1086 Q@Cs, six combinations of three operators (1126 Q@Cs), two combinations of four operators (104 Q@Cs), and exactly one combination of all five operators (6 Q@Cs). For the four DBMSes in our study, the number of generations ranged from five to thirteen.

7.4. Number of Change Points Per Generation

As an illustrative example of how we can look at DBMS query optimizer performance through the lens of change points, let's consider the *maximum number of change points per query*, or *maximum CPQ*, from the perspective of DBMS generations.

For each query, we count the number of Q@Cs at each generation, so for instance a query might have five Q@Cs at generation 1, seven Q@Cs at generation 2, and 17 Q@Cs at generation 5. We then then take the maximum CPQ over the queries (so perhaps 17 is the maximum for generation 5 over all queries). Our hypothesis is that as the generation increases, the maximum query flutter will increase, which then influences the maximum CPQ. The results are shown in Table IV.

Table IV. Max Change Points Per Query, Per Generation

Generation	*Maximum Change Points Per Query*
1	1
2	66
3	115
4	136
5	132
6	141
7	42
8	172
9	2
10	12
11	55
12	29
13	1
Cumulative	172

What we notice is that the maximum CPQ *does* increase with generation, up through generation 8, after which it falls off dramatically. In retrospect, this fall-off makes sense. There can be only 200 Q@Cs for a query, because we look for query plan changes only every 10,000 tuples up to a maximum of 2M tuples. As the generation increases, there is less "room" for more plans (as some fraction of the previously generated plans will still be quite good or even optimal), and so less opportunity for flutter which will show up as a large maximum CPQ. And indeed, at generation 8, there is a query that has an astonishingly high number of change points, 172, all within that generation.

This analysis indicates that there is something concerning that seems to get critical around generation 8. And indeed, for this data set, we'll see in Section 7.7 that the efficacy of the query optimizer bottoms out around that generation.

7.5. Using Change Points

We now consider how to use data already collected, that is the *change points* discussed in Section 6.3: adjacent cardinalities (10K apart; 300 for one DBMS) with different plans. Specifically, how can change points be used to evaluate the effectiveness of different generations of a DBMS?

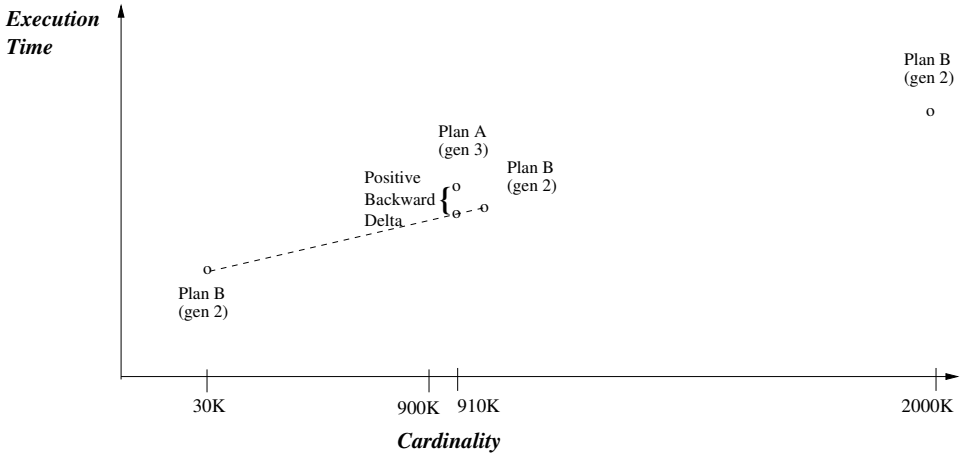


Fig. 9. An Example of a Positive Backward Delta

Define $ops(p)$ for a given plan p to be the set of operators present in that plan, with some operators perhaps repeated in that plan. A generation g (also a set of operators) is *applicable* to a plan p if $ops(p) \subseteq g$. By definition, if a generation is applicable to a given plan, it is applicable to all subsequent generations. The smallest such generation is termed the *minimally applicable generation*, or *mingen*. As change point consists of a pair of adjacent Q@Cs, we have two generations to consider. Lets start with pairs for which $mingen(lower) > mingen(upper)$. An example is shown in Figure 9. In this case, the set of operators in Plan A for the Q@C at cardinality 900K requires at least generation 3, whereas the set of operators for Plan B for the Q@C at cardinality 910K requires but generation 2. (Note that the generations in this example are consecutive, but that is not required. Sometimes the generations within a pair are wildly different.)

To examine the wisdom of picking Plan A for 900K (whose time was measured as the *higher* point shown (the one above the dashed line), we find the closest Q@C also having Plan A. There are two illustrated here, at cardinalities 30K and 2000K, respectively, with the closer one at 30K. (If there is no such Q@C, we can't do the extrapolation, and simply remove that change point from further consideration.)

We use the slope of the dashed line between the measured query time of Plan B at cardinalities 30K and 910K to get an estimated query time at 900k. This realizes an estimate of how fast the query using Plan B would have run on the database having the variable table with a cardinality of 900K. In this particular case, Plan B looks like it would have run *faster* (in fact, was was faster even at 910K in this particular example).

As illustrated in Figure 7, plans can be discontinuous, an example being external merge sort. For that operator, as the cardinality increases, the number of passes will at some point by incremented, but the time is linear for a set number of passes. Such behavior will, for plan B present at the two granularities of 30K and 910K, slightly overestimate the runtime of that plan at 900K, and thus *underestimate* the penalty of going with Plan A rather than Plan B. That said, given that we are extrapolating from the actual run time at a very close cardinality, our extrapolated estimate should be close.

We term this situation a *backward extrapolation*, because we are extrapolating backward from a cardinality of 910K to one of 900K.

With this extrapolation, we can compute the *relative delta*, defined as the measured time of Plan A (the chosen one) at the lower cardinality (900K) minus the extrapolated time of Plan B at that same cardinality, divided by the measured time of Plan A. The relative delta is thus scaled by original measured time at that granularity, and thus has a maximum value of 1.

In this case, the relative delta is positive (the measured time for the plan associated with the higher generation is greater than the extrapolated time), which indicates that the optimizer chose the *wrong plan*: here, Plan B should have been chosen.

On the other hand, a *negative* relative delta (where the measured time for the plan associated with the higher generation is lower than the extrapolated time, such as that illustrated in Figure 8) implies that the additional operator(s) available to the minimally applicable generation of the upper plan were indeed beneficial, in that that plan was faster. (In such cases, we again divide by the larger value, so the minimum is -1.)

Summarizing, a positive relative delta (extrapolated in either the forward or backward direction) implies the presence of suboptimality: the wrong plan was chosen by the DBMS generation, perhaps due to the greater number of operators available. A negative relative delta (forward or backward) implies the *absence* of suboptimality: the chosen plan was faster at that cardinality than the one chosen at the adjacent cardinality. (Note that this is a slightly more expansive definition of suboptimality than that used earlier and illustrated in Figure 1.)

There are six orthogonal possibilities for each change point (that is, a pair of adjacent Q@Cs), a total of 66,769 pairs/change points from the Exploratory and Confirmatory Experiments. (There is also the case of a query instance containing a lone Q@C, which we don't consider further.)

(1) The pair of Q@Cs share the same generation: $mingen(lower) = mingen(upper)$ (62,903 Q@C pairs, 94.2%), which we don't consider further.
(2) The extrapolation yielded a computed query time that was negative, which we also drop (10 pairs, 0.01%).
(3) The extrapolation from above and indicating no suboptimality, termed a *negative backward relative delta*, as exemplified in Figure 8, examined earlier (492 pairs, 0.7%).
(4) The extrapolation, termed a *positive backward relative delta* and exemplified in Figure 9, indicates a suboptimal plan (583 pairs, 0.9%).
(5) The extrapolation was a *negative forward relative delta*, indicating no suboptimality (1218 pairs, 1.8%).
(6) A extrapolation was from below (consider Figure 8 but with Plan A from generation 5; we would then need to extrapolate from the closest Plan B, which is *at a smaller cardinality*), in a *forward direction*, to compute a *positive forward relative delta*, indicating a suboptimal plan (795 pairs, 1.2%).

From these six possibilities, we thus retain (3) and (5), which indicate a suboptimal plan at the higher generation (1710 pairs), and (4) and (6), which indicate a non-suboptimal plan at the higher generation (1378 pairs).

7.6. Trends Across DBMS Generations

The analysis in the previous section is for *a single pair of adjacent Q@Cs for a single query running on a specific DBMS*, providing a *relative delta* for the minimally applicable generation. A positive relative delta indicates that the later generation chose a suboptimal plan; a negative relative delta indicates the later generation did not. The relative delta is a percentage difference, and so is not affected by the absolute magnitude of the run time nor by the cardinality in question. Indeed, because it is a

percentage difference, the relative delta is not affected by the query nor even which DBMS is involved. We associate this relative delta with the later generation, for it is that generation which had the choice between the two plans for that query.

Consider how the *average* relative delta might behave across successive generations. The average relative delta is a characterization of the aggregate impact of that generation, providing a quantitative estimate of the benefit of adding that operator. (We use average so that each point is not impacted by the number of pairs over which that point is computed.) To emphasize, the average aggregate delta aggregates over change points, queries, and even DBMSes, providing a single average relative delta for each generation.

What does our causal model in Figure 5, supported by the correlational and regression analyses of the Confirmatory Experiment in Sections 6.5 and 6.6, say about this? That causal model asserts that as the number of operators increases in subsequent, the latent measures in plan space complexity increase, which therefore impacts suboptimality, also in a positive direction.

Our Gedanken experiment in Section 7.1 takes this behavior and predicts that as the generations contain successively greater numbers of operators, suboptimality will also increase.

If we plot the performance of the DBMS on the y-axis, say as the total time for a workload consisting of a set of queries over a prescribed data set, for a sequence of *DBMS generations* listed on the x-axis, the average relative delta is in some way a characterization of the *slope* of this relationship. A negative average relative delta implies that the indicated generation is doing a good job, with less suboptimality, and so the total workload execution time will go down; a positive average relative delta implies that the suboptimal decisions are dominating, indicated by the total workload execution time going up for that generation.

We expect that the average relative delta for the first few generations will be negative, reflecting new operators that improves some plans, a natural result of the efforts of DBMS developers to increase performance over successive generations of their DBMS. With a negative slope, the performance plot will decrease over successive DBMS generations.

That said, all is not rosy in this picture. Each new operator is applicable to a successively smaller portion of the queries, and perhaps over a successively smaller portion of the cardinality space. As already noted, our causal model predicts that the prevalence of suboptimality would increase as the number of operators available increased. It seems that even with DBMS implementers doing smart things, these two considerations predict that the average relative delta (itself a slope of the performance curve) will *increase*, as suboptimality (a positive relative delta) becomes more prevalent.

This analysis suggests then that the performance curve will thus start to level off. That raises the possibility of the performance curve either asymptotically approaching a horizontal line (the first derivative approaching 0), or worse: the first derivative (the average relative delta) changing to positive, with the average time over the queries we are studying actually *increasing* with DBMS generation.

So the question comes down to this: *by how much* is the increase in efficiency enabled by a new operator (for those Q@Cs utilizing a plan containing that operator) greater than the decrease in efficiency resulting from suboptimality (for a subset of those Q@Cs) resulting from adding that operator? Is there a point where the decrease due to suboptimality obviates the increase enabled by the new operator: a DBMS generation where the average relative delta becomes positive? How close might modern DBMSes be to that limit?

7.7. Have Modern DBMSes Hit The Wall?

We partition the relevant four sets of change points discussed in the Section 7.5 into two groups: (i) those with a *positive* relative delta (either forward or backward), denoting a *suboptimal decision* by the query optimizer at the later generation (corresponding to a higher generation number) and (ii) those with a *negative* relative delta (either forward or backward) relative delta, indicating a *non-suboptimal decision* by the query optimizer at the later generation. (We can't state unequivocally that the plan is optimal, because there may be yet another plan involving the operators within that generation that is even faster.)

Let's first examine those change points for which the query optimizer made a good decision, the non-suboptimal change points; see Table V.

The first thing to note is that this data aggregates the results over the four DBMSes, which had generations ranging from five to 13. Thus this is the first study we are aware of that compares the generational trends of DBMSes over quite disparate code bases, thus getting at fundamental trends.

Overall, only a small number of change points, 1378, or about 2% of the total, satisfied the requirements listed above, including having a different generation number for the lower and upper plans. In fact, there were no such change points for the first three generations nor for the last generation. The second column states the number of change points added by that generation and the average relative delta across just those change points. The last column states the *cumulative relative delta* for that generation, defined as the sum of the average relative delta for those change points associated with plans having only operators in its generation, divided by the number of change points. This is the relevant number for a generation, as it includes all plans that would have been emitted by that generation, using the operators at that generation's disposal.

While the numbers jump around quite a bit, especially for the first few generations. The thing to focus on is the last column, where the cumulative relative delta starting off at a low of -0.28 (recall that low negative value is good, as it indicates the additional operator was effective at lowering the execution time) and slowly decreasing to -0.23: larger negative numbers trending to lower negative numbers.

This general behavior matches our prediction arising from the structural causal model that it gets harder to squeeze out performance gains as operators are added to the DBMS over successive generations.

We now examine those change points for which the query optimizer made a poor decision, in that we can surmise that there was a better plan (the one right next to it in the adjacent Q@C pair). Table VI provides the same information across the DBMS generations for the suboptimal change points: those for which the relative deltas are positive, indicating a poor decision, as the query time increased. While there are still only a small number of change points, there are a greater number of *suboptimal* change points, with more relatively showing up at more recent generations. But more strikingly, the cumulative relative delta has the opposite behavior to the non-suboptimal change points: it *increases* over the generations, starting at 0.08 and ending at 0.25, a value *higher* than that of the non-suboptimal change points.

The next-to-last column brings the non-suboptimal and suboptimal together, stating the *net cumulative relative delta*, gathering all of the change points with plans that would have been generated by that DBMS generation together. We see that the net starts out at -0.24, indicating that the new operators are *decreasing* the query time for the workload. Unfortunately, this happy situation starts to deteriorate: with each successive generation, the improvement is less. This is as predicted: the optimizer is struggling with both more options (plans over the available operators) to select from and a diminished opportunity to make a significant improvement.

Table V. Non-Suboptimal Change Points

Generation	*Number of Change Points*	*Average Relative Delta*	*Cumulative Relative Delta*
1	—	—	0.00
2	—	—	0.00
3	—	—	0.00
4	24	-0.28	-0.28
5	282	-0.118	-0.130
6	309	-0.274	-0.203
7	336	-0.317	-0.243
8	43	-0.20	-0.241
9	4	-0.1	-0.241
10	17	-0.29	-0.241
11	357	-0.178	-0.225
12	6	-0.2	-0.225
13	0	—	-0.225
Cumulative	1378	—	-0.225

Table VI. Suboptimal Change Points

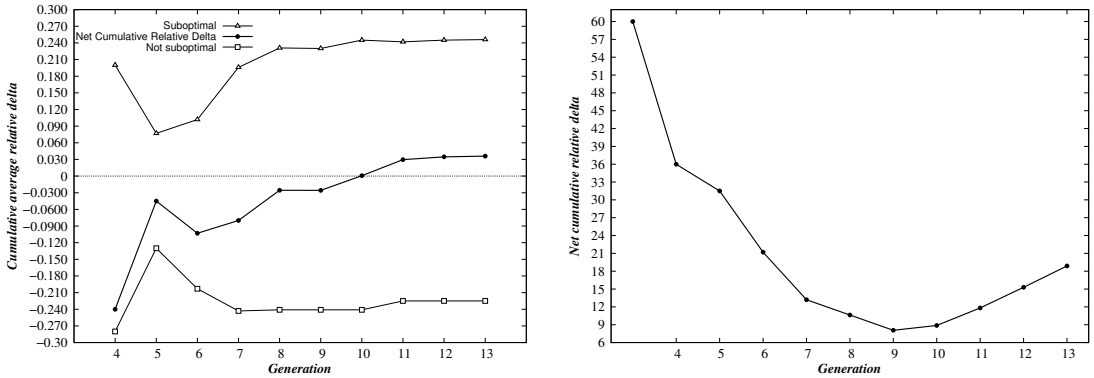
Generation	*Number of Change Points*	*Average Relative Delta*	*Cumulative Relative Delta*	*Net Cumulative Relative Delta*	*Relative "Performance"*
1	—	—	0.00	—	—
2	—	—	0.00	—	–
3	—	—	0.00	—	0
4	2	0.2	0.2	-0.24	-0.24
5	211	0.076	0.077	-0.045	-0.285
6	86	0.16	0.102	-0.103	-0.308
7	263	0.303	0.196	-0.0800	-0.388
8	273	0.303	0.231	-0.0256	-0.401
9	3	0.04	0.230	-0.0257	-0.426
10	168	0.319	0.245	0.008	-0.426
11	639	0.237	0.242	0.0296	-0.396
12	53	0.35	0.245	0.0346	-0.361
13	12	0.40	0.246	0.0360	-0.325
Cumulative	1710	—	0.246	0.0360	—

The final column integrates the average relative delta to produce a unitless *relative performance*, a simulation of how the four DBMSes together would have performed (say, total time) on the workload of the COnfirmatory Experiment, or rather the 1378 change points selected by the criteria in Section 7.5. (We emphasize that our experiment estimates just the first derivative of the performance graph, and then only for a small number of Q@Cs, and then only as a relative measure, between -1 and 1. It is thus only roughly indicative of the *shape* of the performance graph.) We see that at Generation 10, the net actually becomes very slightly positive: meaning that the performance curve has hit a minimum and is now on its way up, towards slower performance. This trend increases in later generations.

Figures 10(a) and 10(b) tell this story graphically, with the net benefit in the center, having a least squares slope of -0.024, which is consistent with the transition from a generation being net beneficial to actually being slightly suboptimal around generation 10 and more so at later generations.

This highly aggregated result, extracting 3000-odd change-point pairs having the specified properties of interest stated in Section 7.6 and drawn from almost 100-thousand Q@Cs implies that these four particular DBMSes, taken together, might be close to or have already hit the wall, where adding an operator actually slows down the average query.

And we recall the analysis in Section 7.4 in which a related problem cropped up around generation 8.



(a) Cumulative Average Relative Delta over Generations (b) "Performance" over Generations

Fig. 10. Trend of Performance Improvement over Generations

It is important to emphasize that we can't say whether any of these DBMSes have actually transitioned to where, for this class of queries, the errors of the suboptimal change points overwhelm the benefits of the non-suboptimal change points. Presumably, the DBMS vendors have done extensive tests to ensure that the operator added to each generation did in fact effect a speedup on the representative workloads that they use in evaluating their optimizer enhancements. However, we *can* say that (a) the trends observed strongly point to a decreasing benefit and an increasing cost. as predicted by the simple arguments made above, and (b) if current DBMSes haven't yet reached the point of diminishing returns, that possibility exists.

We should also reiterate all the provisos mentioned earlier. In all of these experiments, we are looking at quite simple queries, over a quite limited range of data, with only a small percentage (3%) of change points identified in our experimental protocol. On the other hand, simpler queries are thought to be easier to generate good plans than more complex queries (as Hypothesis 2 in Section 4.3 states), relational data is generally much less uniform in its structure (e.g., we used only integer data), values (the values in our tables are quite evenly distributed), schema (the schemas of our tables are identical and extremely simple), and range of sizes (the smallest table is 1/200-th of the largest table); all of these factors should independently, and certainly in concert, minimize the suboptimality.

8. ENGINEERING IMPLICATIONS

We studied a particular phenomenon, suboptimality, when the optimizer selecting a wrong plan. This phenomenon is indicated by the existence of a query plan that performs more efficiently than the DBMS's 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. The genesis of our predictive model was a sense that suboptimality is caused in part by the inherent complexity of these system and the concomitant unanticipated interactions between various rules in the optimizer.

Through a series of experiments managed by our laboratory instrument management system, DBLAB, carried out across several years, 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 quite simple queries.
— A quarter of the queries exhibited significant suboptimality ($\geq 20\%$ 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.

— Many queries exhibited *query fluttering*, in which the query optimizer returned to a previous plan at a higher cardinality.
— Some queries exhibited significant *query thrashing*, with a plan change at almost every cardinality. While this phenomenon was first visualized by Haritsa et al. [8; 9] 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 right-hand 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. **TODO:** *Rick: check with Sabah's results and modify around here if needed.*

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 [23]), 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 in Section 3 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; 16]. 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).

The generational study in Section 7 though implies that the challenge in daunting. That study validates the implications of the causal model in Figure 5, which correctly predict the almost inexorable rise in net cumulative relative delta, which implies that the suboptimizer will eventually hit the wall where it is no longer improving.

We emphasize that this section of this paper, considering engineering implications of the underlying causal model, contrasts with the rest of the paper, whose focus is on the science and on understanding these complex systems at a fundamental level. Good engineering should be built on solid scientific results. This paper focuses on the latter.

9. 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. **TODO:** *Rick:make consistent with list at the beginning of the paper*

— Shows that even for simple queries, over a simple schema and relatively small range of sizes, the prevalence of *query suboptimality*, *query flutter*, and *query thrashing*, three problems that have not been systematically investigated across DBMSes, is high, and thus there is still research needed on this mature topic of query optimization.
— Introduces a new *methodological perspective* that treats DBMSes as experimental subjects within *empirical generalization*.
— 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 *seven hypotheses* deductively derived from the predictive 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 key attributes, moderates these three interactions. **TODO:** *Rick:Check again*
— 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 **TODO:** *Rick: state. Do we want to keep the following?* . 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.

— Articulates for the first time a possible *upper bound* on the number of operators a DBMS may be able to support, given that the empirical evidence suggests that additional operators speed up a smaller and smaller portion of the query/cardinality space while incurring an increasing chance of suboptimality over the remaining space, which is growing.
— Applies a novel experiment over pairs of adjacent Q@Cs to show that such an upper bound is probably and may have already been reached with one or more our our subject DBMSes.
— Identifies *specific directions for engineering interventions*.
— 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.

This paper thus suggests a framework of casual model elaboration and directed engineering efforts.

10. FUTURE WORK

There are several directions this work could be take. **TODO:** *RICK IS HERE* With the model refinements we propose here, additional directed pointers to engineering efforts should emerge. **TODO:** *Rick:DOn't say "we", but just say the research needs to be done*

TODO: *Rick: keep following shorten?*

We want to look into query flutter and thrashing in greater detail, as those phenomena provide concrete indicators of problematic optimizer behavior. One possible methodological approach is to utilize SQL optimizer hints, such as "`+ SEMIJOIN`" in MySQL and "`enable_hashjoin(false)`" in PostgreSQL, to encourage the optimizer to produce more plans at a given cardinality, that can then be timed to make more explicit the entire plan space (recall that CEPS, the cardinality of the effective plan space, is no greater and probably much smaller than the cardinality of the plan space).

The investigations in this paper were based on very simple SPJ (select-project-join) queries. We wish to also consider *schema complexity*: foreign keys and indexes and *query complexity*: complex predicates, subqueries, and user-defined data types, methods, and operators. We also want to manipulate DBMSes internally (at least for those that are open-source), turning on and off the rules and observing suboptimality. 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 suboptimal 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." [16, 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.

TODO: *Rick: revise: talk about generational study, how we're hitting a wall, and how it might be necessary*

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.

Appendix A provides further details on the experiments.

REFERENCES

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

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.

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

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

S. Currim, R. T. Snodgrass, Y. -K. Suh, and R. Zhang, "DBMS Metrology: Measuring Query Time," in *Proceedings of the ACM SIGMOD Conference*, pp. 421–432, June 2013.

S. Currim, R. T. Snodgrass, Y. -K. Suh, and R. Zhang, "DBMS Metrology: Measuring Query Time," ACM *Transactions on Database Systems*, 42+8 pages, October, 2016.

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

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.

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

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

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,

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

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

ISO, "ISO SQL:2008 International Standard," 2008.

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

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

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

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

J. Melton (editor), ISO/IEC 9075, Database Language SQL:2011 Part 2: SQL/Foundation, December, 2011.

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

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.

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

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.

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.

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

A. DETAILS ON THE EXPERIMENTS

Table I lists the run statistics of the seven experiments used in this paper. In this appendix we provide more detailed information on the experiments.

Table VII gives some of those details. We'll walk through the columns in succession.

The third column states the data set used in each experiment, that is, the specific tables being queried. There are four data sets, named A, B, C, and D.

We first discuss the features shared between the four data sets. As introduced in Section 2, the queries referenced tables `ft_HT1`, `ft_HT2`, `ft_HT3`, and `ft_HT4`. All four tables contain four columns, each of type integer. The specific values of the rows for all but the first column depend on the percentage of skewed data. Section 5.4 provides the algorithm for generating the values for different values of skew; this algorithm is used in columns two–four, which for any row will have identical values. The first column holds a unique integer starting from 1 and going to 60K or 2M, for use in an optional primary key.

There was one version of the last three tables, for use with MySQL, with cardinality 60K, and one version for the rest of the DBMSes, with cardinality 2M.

We generate 200 versions of `ft_HT1`, termed the *variable table*. For MySQL, these version contain 300, 600, 900, 1200, ..., 59,700, and 60,000 rows; for the rest of the DBMSes, these versions contain 10,000, 20,000, 30,000, ..., 1,970,000, and 2M rows, as introduced in Section 2.

We now differentiate the data sets, elaborating on the discussion in Sections 6.2–6.3. Data Set C is perhaps the simplest to describe: it specifies no primary key, has no duplicate rows, and has no skew (of course, for any of the four tables). Data Set A differs from Data Set C only in that there is skew. As summarized at the end of Section 5.4, we use five skewness values: 0 (approximately), 0.001, 0.1, 0.5, and 1.0.

Data Set B is similar to Data Set A, adding the specification of the first column as the primary key. And Data Set D is similar to Data Set B, adding the specification that the other three columns should each be associated with a secondary index, only only that one column. We see the confirmatory examined a much larger variation of data sets than the exploratory studies.

The next column of Table VII concerns the *Lab Shelf*. DBLAB utilizes the metaphor of a bookshelf of lab notebooks. Here, each shelf is associated with a version of DBLAB itself. For the experiments in this paper, we used at various times over the last three years lab shelves (that is, program versions) 5.19, 5.2, 6.0, and 7.1. Versions 5.19 and 5.2 were very similar; both implemented TTPv1. Version 6.0 also implemented TTPv1, but collected more query measures that were not relevant for this paper. Version 7.1 implemented TTPv2.

The DBLAB system also includes support for *experiment scenarios*, which are Java code that actually performs the experiment, such as varying the cardinality and running different queries on the data. The only difference in the scenario code across these experiments was in accommodating the details of the data set (that is, creating secondary indexes and data skew) and adding robustness features, such as using exponential backoff when the network connection to the DBMS was temporarily lost.

The bottom line is that while the lab shelf and experiment scenario varied somewhat, the only important aspect was the *Protocol* column of Table I.

We now turn to the fifth column, "what was examined?" Here there are just two possibilities, all 200 cardinalities or just the cardinalities at which the query plan changed. The sixth column, "what was timed?", indicates that experiments three and four, Exhaustive and Exhaustive with Keys, described in Sections 5.5 and 6.3, just collected query plans, and thus took much less time to run.

Table VII. Experiments 1–7: Detailed Run Statistics

	Experiment	*Data Sets Used*	*Lab Shelves*	*What was Examined?*	*Was query Timed?*	*Number of Retained (Raw) Q@Cs*
1	Monotonicity	A	6.0	all cardinalities	yes	12,000 (12,000)
2	Initial Exhaustive	A	5.19 + 6.0	all cardinalities	yes	27,948 (32,000)
3	Exhaustive	A	7.1	all cardinalities	no	29,515 (32,000)
4	Exhaustive with Keys	B	6.0	change points	no	—
5	Initial Exploratory	A + B	5.19 + 5.2 + 6.0	change points	yes	8,171 (8,842)
6	Exploratory	A + B	7.1	change points	yes	12,100 (12,560)
7	Confirmatory	A + B + C +D	7.1	change points	yes	94,502 (99,558)
	Total					184,236 (196,960)

The last column states how many Q@Cs the experiments measured (each with 10 QEs), termed *raw*, and how many Q@Cs were retained after the protocol implied by the fourth column and listed explicitly in the third column of Table I dropped query executions and Q@Cs via its many sanity checks.

TODO: *Rick: I (Young) updated the following. Please review it.* The following sets of queries were used in the seven experiments.

QSa. 100 queries over the four tables, generated as described in Section 5.3
QSb. 100 queries, generated the same way
QSc. 100 queries
QSd. 100 queries
QSe. 100 queries
QSf. 100 queries
QSg. 100 queries
QSh. 100 queries
QSi. 100 queries
QSj. 100 queries
QSl. A primary key query set consisting of 230 queries from *QSa–QSf*
QSm. A primary key query set consisting of 160 queries: 40 queries drawn from each of *QSg–QSj*
QSn. A primary key query set consisting of 110 new queries
QSo. A query set consisting of 100 queries without aggregates (for Data Set C)
QSp. A subquery query set consisting of 100 queries, each with a subquery
QSq. A subquery query set consisting of 100 queries, each with a subquery
QSr. A query set consisting of 390 queries drawn from *QSl* + *QSm*
QSs. A primary key query set consisting of 100 queries drawn from *QSr*

Experiment 1 (Monotonicity) used the first 50 queries from QSa for one DBMS plus the first six queries from QSa and two queries each from *QSd* and *QSe* for MySQL, for a total of 60 query instances.

Experiment 2 (Initial Exhaustive) used the first 50 queries from *QSa* for the three other DBMSes plus the ten queries for MySQL from Experiment 1, for a total of 160 query instances. Experiment 3 (Exhaustive) used the same 160 queries.

Experiment 4 (Exhaustive with Keys) used the first 50 queries from *QSa* for all four DBMSes, for a total of 200 query instances.

Experiment 5 (Initial Exploratory) used the (100) queries from *QSa–QSf* (for one DBMS), the first 20 queries from *QSa–QSf* (for another DBMS), the first 10 (primary key) queries from *QSa* (for all four DBMSes), and the first 10 (non-primary key) queries from *QSa* (for the other two DBMSes), for a total of 780 query instances.

Experiment 6 (Exploratory) used *QSa* and *QSb*, plus the first 100 (primary key) queries from *QSr*, for all four DBMSes, for a total of 1200 query instances.

Experiment 7 (Confirmatory) used *QSc–QSj*, along with *QSr* except the (first 100) queries included in Experiment 6, *QSn* for primary key (for two runs, or 220 queries), *QSo* for no data skew, *QSp* for primary key and subquery, *QSp* for subquery, *QSp* and *QSq* for primary key and secondary index and subquery, *QSs* for primary key and secondary index, all across all four DBMSes, for a total of 7,640 query instances.