Xplus: A SQL-Tuning-Aware Query Optimizer

Herodotos Herodotou and Shivnath Babu*
Department of Computer Science
Duke University
{hero,shivnath}@cs.duke.edu

ABSTRACT

The need to improve a suboptimal execution plan picked by the query optimizer for a repeatedly run SQL query arises routinely. Complex expressions, skewed or correlated data, and changing conditions can cause the optimizer to make mistakes. For example, the optimizer may pick a poor join order, overlook an important index, use a nested-loop join when a hash join would have done better, or cause an expensive, but avoidable, sort to happen. SQL tuning is also needed while tuning multi-tier services to meet service-level objectives. The difficulty of SQL tuning can be lessened considerably if users and higher-level tuning tools can tell the optimizer: "I am not satisfied with the performance of the plan p being used for the query Q that runs repeatedly. Can you generate a (δ%) better plan?" This paper designs, implements, and evaluates *Xplus* which, to our knowledge, is the first query optimizer to provide this feature. Xplus goes beyond the traditional plan-first-execute-next approach: Xplus runs some (sub)plans proactively, collects monitoring data from the runs, and iterates. A nontrivial challenge is in choosing a small set of plans to run. Xplus guides this process efficiently using an extensible architecture comprising SQL-tuning *experts* with different goals, and a *policy* to arbitrate among the experts. We show the effectiveness of Xplus on real-life tuning scenarios created using TPC-H queries on a PostgreSQL database.

1. INTRODUCTION

Database query optimizers have predominantly followed a *plan-first execute-next* approach [22]: the optimizer uses a search algorithm to enumerate plans, estimates plan costs based on a performance model and statistics, and picks the plan with least estimated cost to execute a given SQL query. While this approach has been widely successful, it causes a lot of grief when the optimizer mistakenly picks a poor plan for a query that is run repeatedly (e.g., by a business intelligence or report generation application).

Optimizer mistakes are common in practice: an important index may not be used in the selected plan, the choice of join order may be poor, or an expensive, but avoidable, sort may get done [10]. A database administrator (DBA) or expert consultant may have to step in to lead the optimizer towards a good plan [6]. This process of improving the performance of a "problem query" is referred to in the database industry as *SQL tuning*. Tuning a problem query is critical in two settings:

- *Bad plan setting:* Cardinality (number of tuples) estimation errors due to unknown or stale statistics—common for complex queries over skewed or correlated data—can cause optimizers to pick a highly suboptimal plan for a query.
- *SLO setting:* Databases are used as one of the tiers in multi-tiered services that need to meet *service-level objectives (SLOs)* on response time or workload completion time (e.g., all reports should be generated by 6.00 AM). Tuning a multi-tiered service to meet SLOs creates SQL tuning tasks (e.g., find a plan that can execute query Q in under 10 minutes).

Need for a SQL-Tuning-Aware Query Optimizer: SQL tuning is a human-intensive and time-consuming process today, and expensive in terms of the total cost of database ownership. The pain of SQL tuning can be lessened considerably if query optimizers support a feature using which a user or higher-level tuning tool can tell the optimizer: "I am not satisfied with the performance of the plan p being used for the query Q that runs repeatedly. Can you generate a (δ%) better plan?" This paper designs, implements, and evaluates *Xplus* which, to the best of our knowledge, is the first query optimizer to provide this feature.

The key to effective tuning of Q is to make the best use of the information available from running a plan/subplan for Q; and to repeat this process until a satisfactory plan is found. The following information is available for each operator O of a plan p after p runs:

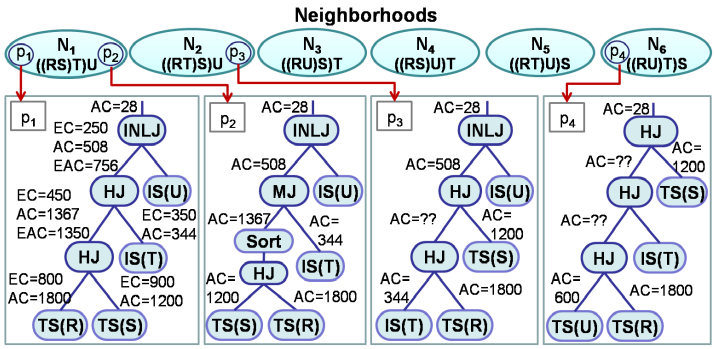
- *Estimated Cardinality (EC):* the number of tuples produced by O as estimated by the optimizer during plan selection.
- *Actual Cardinality (AC):* the actual number of tuples produced.
- *Estimated-Actual Cardinality (EAC):* the number of tuples produced by O as estimated by the optimizer if the optimizer knew the actual cardinality (AC) values of O's children. (EAC is a new concept introduced in this paper.)

Existing Approaches: The *Learning Optimizer (Leo)* [23] incorporates AC values obtained from previous plan runs to correct EC values during the optimization of queries submitted by users and applications [9]. The *pay-as-you-go* approach takes this idea further using proactive plan modification and monitoring techniques to measure approximate cardinalities for subexpressions to supplement the AC values collected from operators in a plan [8]. The overall approach of [23, 8, 9] has some limitations in practice:

- *Risk of unpredictable behavior:* Making changes to plans of user-facing queries runs the risk of performance regression because incorporating a few AC values alongside EC values can sometimes lead to selection of plans with worse performance than before [19]. DBAs and users usually prefer predictable,

*Supported by NSF awards 0917062 and 0964560

Proceedings of the VLDB Endowment, Vol. 3, No. 1



Neighborhood Table

ID	Cardinality Set	Covered	Least-Cost Plan
N_1	R, S, T, U, σ_p(R), σ_p(S), σ_p(R)⋈σ_p(S), σ_p(R)⋈σ_p(S)⋈T, σ_p(R)⋈σ_p(S)⋈T⋈U	Yes	p_1
N_2	R, S, T, U, σ_p(R), σ_p(S), σ_p(R)⋈T, σ_p(R)⋈σ_p(S)⋈T, σ_p(R)⋈σ_p(S)⋈T⋈U	No	p_3
⋮			
N_6	R, S, T, U, σ_p(R), σ_p(S), σ_p(R)⋈U, σ_p(R)⋈U⋈T, σ_p(R)⋈σ_p(S)⋈T⋈U	No	p_4

Cardinality Table

Expression	EC	AC
σ_p(R)	800	1800
σ_p(S)	900	1200
σ_p(R)⋈σ_p(S)	450	1367
σ_p(R)⋈T	650	??
σ_p(R)⋈σ_p(S)⋈T	250	508
σ_p(R)⋈U⋈T	320	??
⋮		
σ_p(R)⋈σ_p(S)⋈T⋈U	12	28

Figure 1: (a) Neighborhoods and physical plans, (b) Neighborhood Table, and (c) Cardinality Table for our example star-join query

possibly lower, performance over unpredictable behavior [4].

- *Imbalanced use of exploitation and exploration:* Effective tuning of a problem query needs to balance two conflicting objectives. Pure exploitation recommends running the plan with least cost based on the current cardinality estimates. Leo and pay-as-you-go take this route which ignores the uncertainty in estimates when picking the plan for the query. In contrast, pure exploration recommends running the plan whose execution will produce information that reduces the uncertainty in current estimates the most, while ignoring plan costs. Oracle's *Automatic Tuning Optimizer (ATO)* [6] takes an exploration-oriented route where base tables and joins are first sampled to reduce the uncertainty in their cardinality estimates.
- *No support for the SLO setting:* No current optimizer supports the SLO setting where systematic exploration can be critical.

Xplus addresses these limitations. A user or tuning tool can mark a query Q for which the performance of the plan being picked is not satisfactory; and Xplus will try to find a new plan that gives the desired performance. If Xplus fails to find such a plan, it will still produce Q's optimal plan for the current database configuration and optimizer performance model; with all plans costed using accurate cardinalities. Xplus works with the existing database configuration. While configuration changes (e.g., new indexes, changes to server parameters, or provisioning more resources) can also improve Q's performance, such changes are disruptive in many ways including changes to the performance of other queries. If Xplus fails, then we have the strong guarantee that disruptive changes are unavoidable to get the desired performance for Q. Appendix A gives a detailed comparison of Xplus with other SQL tuning approaches.

We introduce a running example to illustrate the challenges faced during SQL tuning. Suppose Xplus has to tune a star-join query Q that joins four tables R (fact table), S, T, and U, with filter conditions on R and S. Figure 1(a) shows four valid execution plans p_1-p_4 for Q. Let p_1 be the least-cost plan picked by the query optimizer. Figure 1(a) shows the EC, AC, and EAC values for each operator in plan p_1 that are available after a run of p_1.

- *Cardinality estimation errors:* A large difference between an AC and corresponding EC value indicates a cardinality estimation error. For example, AC=1800 differs significantly from EC=800 for the filtered table scan over R, caused possibly by unknown correlations among columns of R.
- *Error propagation:* Blind comparison of AC and EC values can lead to wrong conclusions because estimation errors propagate up the plan. This issue can be addressed by comparing AC values with corresponding EAC values. For example, consider the hash join between R and S in plan p_1. While the gap between AC=1367 and EC=450 is large for this join, the EAC=1350 is close to AC. (This EAC was computed based on the actual cardinalities, 1800 and 1200, of the join's children.) This information suggests that the estimation error in the join's output cardinality is almost exclusively due to wrong input cardinality estimates, rather than a wrong join selectivity estimate.
- *Livelock in SQL tuning:* Suppose p_2 is the new least-cost plan when the AC values from p_1 are incorporated with existing statistics. Approaches like Leo will use p_2 to run the next instance of Q submitted, which is problematic. Although p_2 is seemingly very different from p_1, p_2 will not bring any *new* AC values. Thus, no further progress will be made towards finding better plans for Q; we say that the tuning of Q is *livelocked*.[1]
- *Use of EAC:* Comparing EAC and AC values for joins indicates which joins are more (or less) selective than the optimizer's estimate. For example, R's join with T, whose EAC=756 > AC=508, turned out to be much more selective than estimated. Such information can guide whether a plan like p_3, which has a different join order from p_1, could improve performance. A run of p_3 will also bring the unknown AC value for $R \bowtie T$.
- *Exploitation Vs. exploration:* Would plan p_4 be preferable to p_3 as the plan to run next because a run of p_4 will bring two unknown AC values compared to just one from p_3? At the same time, p_4 is riskier to run because its cost estimate relies on two unknowns. This issue is the manifestation in SQL tuning of the classic "exploit or explore" dilemma in machine learning [12].
- *Efficiency features:* Running a subplan instead of the full plan will often bring all the *new* AC values that the plan brings, e.g., the $R \bowtie T$ subplan will bring all new AC values for p_3.

Our Contributions: As this example shows, a number of novel opportunities and challenges await a SQL-tuning-aware optimizer like Xplus. We make the following contributions in this paper:

- Xplus uses a novel abstraction called *plan neighborhoods* to capture useful relationships among plans that simplify information tracking and improve efficiency in SQL tuning.
- Xplus balances exploitation and exploration efficiently using a combination of tuning *experts* with different goals, and a *policy* to arbitrate among the experts. Efficiency is further improved through features like subplan selection and parallel execution.
- Xplus enables the tuning overhead on the production database to be bounded. Xplus is also designed to leverage recent solutions that let the database system run query plans noninvasively in sandboxed [6] and standby [6, 11] settings for tuning.
- The Xplus architecture emphasizes modularity as well as usage as a standalone tuning tool while being merged incrementally into commercial optimizers that are very complex software.
- We report an extensive evaluation of Xplus based on SQL tuning scenarios that arise routinely in practice.

[1]Livelocks are well-studied phenomena where a process appears to be executing, but makes no real progress towards its goal.

2. FOUNDATIONS OF XPLUS

2.1 Inputs and Output of a Tuning Session

Xplus can be used both as a regular optimizer—given a query, return the plan with least-estimated cost—and as a tuning optimizer where it runs *tuning sessions*. A tuning session takes three inputs:

1. The query Q and its current plan p to be tuned.
2. The *stopping condition* to stop tuning Q. Current options are to run Xplus until (i) a new plan is found that is a given $\delta\%$ better than p; or (ii) a given time interval; or (iii) Xplus can cost all plans for Q using accurate cardinalities (a novel contribution); or (iv) the user exits the tuning session when she is satisfied with the performance of the best plan found so far.
3. *Resource constraints* to limit the overhead placed by Xplus on the regular database workload. Xplus accepts a parameter MPL_T (multiprogramming level of tuning) that represents the maximum number of plans Xplus can run concurrently.

Xplus outputs the best plan p' found so far in the tuning session, and the improvement of p' over p. Database systems provide multiple ways to enable p' to be used for future instances of Q: (i) associating a plan with a query template (*stored outlines* in Oracle [6], *abstract plans* in Sybase ASE [1], and *Explain_Plan* in PostgreSQL [15]), (ii) *optimizer hints* in Microsoft SQL Server [7] and IBM DB2 [17], and (iii) *optimization profiles* [16] in DB2.

2.2 Plan Neighborhoods

We begin by discussing the plan neighborhood abstraction that Xplus uses to partition a query Q's physical plan space. The cost-based optimizer in a database system will use a *performance model* to estimate the cost of plans for Q. The performance model consists of a *cardinality model* and a *cost model* [14]. The cardinality model is used to estimate the cardinality of relational algebra expressions defining the data processed as input and produced as output by each operator in a plan. Given these cardinality estimates, the cost model estimates the execution cost of each operator. The cost model takes as input factors such as CPU and disk-access speeds, memory availability, and data layout on disk. While modern cost models have been validated to be quite accurate when cardinalities are known, the cardinality models are laced with simplifying assumptions that can introduce order-of-magnitude errors [14].

Definition 1. **Cardinality Set:** To estimate the execution cost of an operator O in a plan p, the optimizer uses its cardinality model to estimate cardinalities for a set *CS(O)* of relational algebra expressions. *CS(O)* is called O's cardinality set. The cardinality set *CS(p)* of plan p is the set union of the cardinality sets of all operators in p.

A function *Generate_CS(O,p)* is written per physical operator type to return the cardinality set *CS(O)* of an instance O of that type in a plan p. The expressions in *CS(O)* are a subset of the relational algebra expressions that define each of O's inputs and outputs in p. (Appendix B.1 discusses the *Generate_CS(O,p)* functions for the physical operators in PostgreSQL.) *CS(p)* is generated by invoking the *Generate_CS(O,p)* functions of all operators in plan p using a bottom-up plan traversal.

Definition 2. **Plan Neighborhood:** The space of physical plans for a query Q can be represented as a graph G_Q. Each vertex in G_Q denotes a physical plan for Q. An edge exists between the vertices for plans p_1 and p_2 if *CS(p_1)* $\subseteq$ *CS(p_2)* or *CS(p_2)* $\subseteq$ *CS(p_1)*. The connected components of G_Q define the plan neighborhoods of Q.

CS(N), the cardinality set of plan neighborhood N, is the set union of the cardinality sets of all plans in N, i.e., the cardinalities needed to calculate execution costs for all plans in N. Section 3.1 discusses the enumeration of plan neighborhoods.

Xplus uses two data structures to store all information about neighborhoods: *Neighborhood Table* and *Cardinality Table*. Figures 1(b) and 1(c) illustrate these two data structures for our running example query from Section 1. The Neighborhood Table stores information about all neighborhoods—including cardinality set and current least-cost plan—with each row corresponding to one neighborhood. Each row in the Cardinality Table stores the EC and (if available) AC values of a relational algebra expression needed for costing. The initialization, use, and maintenance of these data structures during a tuning session are discussed in Section 3.

Figure 1(b) shows the cardinality set of neighborhood N_1 for our running example. Plans p_1 and p_2 from Figure 1(a) belong to N_1, with *CS(p_1)*$\subset$*CS(N_1)* and *CS(p_2)*$\subset$*CS(N_1)*. We get *CS(p_1)*=*CS(p_2)* even though p_2 uses a merge join, has an extra sort, and a different join order between R and S compared to p_1. Plans p_3 and p_4 belong to different neighborhoods than N_1 since *CS(p_3)* and *CS(p_4)* are disjoint from *CS(N_1)*.

2.3 Coverage of Neighborhoods

The progress of Xplus while tuning a query Q can be described in terms of the *coverage* of neighborhoods in Q's plan space.

Definition 3. **Covering a Neighborhood:** A neighborhood N is covered when AC values are available for all expressions in *CS(N)*.

When a tuning session starts, only the AC values from the plan picked originally for Q by the optimizer may be available. More AC values are brought in as Xplus runs (sub)plans during the tuning session, leading to more and more neighborhoods getting covered.

Property 1. Once a neighborhood N is covered for a query Q, Xplus can guarantee that all plans in N are costed with accurate cardinalities. Xplus can now output the optimal plan in N for the given database configuration and optimizer cost model. □

Property 2. Once all neighborhoods are covered for a query Q, Xplus can output Q's optimal plan for the given database configuration and optimizer cost model. □

The efficiency with which Xplus provides these strong guarantees is captured by Properties 3-5.

Property 3. Xplus runs *at most* one (possibly modified) plan in a neighborhood N to obtain AC values for *CS(N)*. □

Section 3.3 presents techniques for plan modification. Property 3 allows Xplus to reduce the number of plans run to tune a query by maximizing the use of information collected from each plan run.

Property 4. Xplus can cover all neighborhoods for a query Q by running plans for only a fraction of Q's neighborhoods. Almost all these runs are of subplans of Q (and not full plans). □

Consider our running example star-join query and plan p_1 selected originally by the optimizer (Figure 1(a)). Xplus can fully tune this query to provide the strong guarantee in Property 2 by running only two subplans: one in neighborhood N_3 and one in N_5. As a real-life example, PostgreSQL has around 250,000 valid plans for the complex TPC-H Query Q_9; which gave 36 neighborhoods. Xplus only ran 8 subplans to fully tune this query and give a 2.3x speedup compared to the original PostgreSQL plan.

Property 5. Xplus can determine efficiently the minimum set of neighborhoods that contain plans whose cost estimates will change based on AC values collected from running a plan for a query. □

3. HOW XPLUS WORKS

We will now discuss how Xplus enumerates neighborhoods and plans, chooses the next neighborhood to cover at any point, and selects the (sub)plan to run to cover the chosen neighborhood.

3.1 Enumerating Neighborhoods and Plans

Definition 2 lends itself naturally to an approach to enumerate all neighborhoods for a query Q: (i) enumerate all plans for Q and their cardinality sets; (ii) generate the vertices and edges in the corresponding graph G_Q as per Definition 2; and (iii) use Breadth First Search to identify the connected components of G_Q. This approach quickly becomes very expensive for complex queries, so we developed an efficient alternative based on plan *transformations*.

Definition 4. **Transformation:** A transformation τ when applied to a plan p for a query Q, gives a different plan p' for Q. τ is an *intra (neighborhood) transformation* if p and p' are in the same neighborhood; else τ is an *inter (neighborhood) transformation*.

Given a sound and complete set of transformations for a given database system, Xplus can use:

- Inter transformations to enumerate all neighborhoods efficiently for a query starting from the initial plan given in the tuning session. One plan is stored per neighborhood (see Figure 1(b)).
- Intra transformations to enumerate all plans in a neighborhood efficiently starting from the stored plan.

We have developed a set of transformations applicable to the physical plans of PostgreSQL. Transformations are implemented as functions that are applied to a plan data structure (similar to work on extensible query optimizers, e.g., [13]). The details are given in Appendix B.2.

3.2 Picking the Neighborhoods to Cover

Suppose Xplus were to use an approach like Leo [23] to select which neighborhoods to cover: find the current least-cost plan p_{opt} in the plan space, and run p_{opt} to collect AC values to cover the neighborhood that contains p_{opt}. The new AC values and recosting of plans may lead to a new least-cost plan p'_{opt}. If so, p'_{opt} is run to cover the corresponding neighborhood; and the process repeats. However, this approach will get stuck in a local optimum—what we call a livelock—if the neighborhood containing p'_{opt} is already covered. Thus, no new AC value will result from running p'_{opt}, and the query will not be tuned further. Another serious problem with this approach is that it limits opportunities for efficient parallel processing, but we will leave parallel processing to Section 4.

What is missing in the above approach is the ability to explore the space of physical plans. Exploration will allow the optimizer to collect additional information—thereby converting more cardinality estimates from uncertain to accurate—which eventually could lead to a plan better than the current one. In *pure exploration*, the focus is on reducing the uncertainty in cardinality estimates, so a plan that resolves the current uncertainty the most will be preferred over other plans. In contrast, in *pure exploitation*, the plan with least cost based on current cardinality estimates is always preferred, while the uncertainty in these estimates is ignored. Exploitation and exploration are naturally at odds with each other, but elements of both are required in holistic SQL tuning.

Xplus balances these conflicting objectives using multiple goal-driven *experts*—given the current global state, an expert has its own assumptions and strategy to recommend the next neighborhood to cover—and a *selection policy* to arbitrate among the experts.

3.2.1 Experts

We describe the design of four experts whose combined behavior has worked very well in common query tuning scenarios.[2] These experts operate with different degrees of exploitation and exploration as summarized in Table 1. Implementation details of these experts are presented in Appendix B.3.

Expert	Exploitation Component	Exploration Component	Can run into a Livelock?
Pure Exploiter	Highest	None	Yes
Join Shuffler	High	Low	Yes
Base Changer	Low	High	No
Pure Explorer	None	Highest	No

Table 1: Properties of the current experts in Xplus

Pure Exploiter Expert: The Pure Exploiter simply recommends the neighborhood N_{opt} containing the plan with the lowest estimated cost based on the currently available information. N_{opt} is recommended if it has not been covered yet. Otherwise, the Pure Exploiter is livelocked and has no neighborhood to recommend.

Join Shuffler Expert: The Join Shuffler is one of the current experts in Xplus whose recommendation strategy is a mix of exploitation and exploration. This expert leans more towards exploitation based on the observation that the more selective joins should appear earlier in the plan's join tree. The Join Shuffler works with the current least-cost plan p_{best} among all covered neighborhoods. It uses AC and EAC values in p_{best} to identify joins for which the join selectivity was overestimated, and uses inter transformations to push these joins as low in the plan as possible. If these transformations result in a plan in an uncovered neighborhood N, then N is recommended; otherwise the Join Shuffler is livelocked. For the example tuning scenario from Section 1, the Join Shuffler may recommend neighborhood N_2 because it contains plans like p_3.

Base Changer Expert: The Base Changer's recommendation strategy also mixes exploitation and exploration, but this expert leans more towards exploration. The motivation for this expert comes from the observation that the choice of which two tables to join first (called the *base* join) in a join tree often has a high impact on the overall performance of the tree. As anecdotal evidence, optimizer *hints* in most database systems have direct support to specify the first join or the first table to use in the join tree, e.g., the *leading* hint in Oracle [6]. The Base Changer considers each two-way join in the query as a possible base, and creates the rest of each join tree based on the join order in the current least-cost plan p_{opt}. This strategy causes the Base Changer to recommend neighborhoods with plenty of uncertain cardinality estimates. The mild exploitation component of this expert comes from using p_{opt} to build parts of the join trees. Unless all neighborhoods are covered, the Base Changer will always have a neighborhood to recommend; it will never run into a livelock (unlike the previous two experts).

Pure Explorer Expert: If the overall degree of certainty in cardinality estimates is low, then a lot of information may need to be collected in order to find an execution plan with the desired performance. Compared to the other experts, the Pure Explorer is designed to gather more statistics quickly. Upon invocation, this expert recommends the uncovered neighborhood with the highest number of uncertain cardinality estimates in its cardinality set. Like the Base Changer, the Pure Explorer will never run into a livelock.

3.2.2 Selection Policy for Experts

Xplus supports three different policies to determine which expert's recommendation should be followed at any point of time.

Round-Robin Policy: This policy consults the experts in a round-robin fashion. Apart from its simplicity, this policy has the advantage of ensuring fairness across all experts.

Priority-based Policy: This policy assigns a predefined priority to each expert. Each time a new recommendation is needed, the experts are consulted in decreasing order of priority. If an expert

[2]We expect that the current set of experts in Xplus will be refined and extended as Xplus is tried in different environments. The architecture of Xplus makes it easy to add and drop experts.

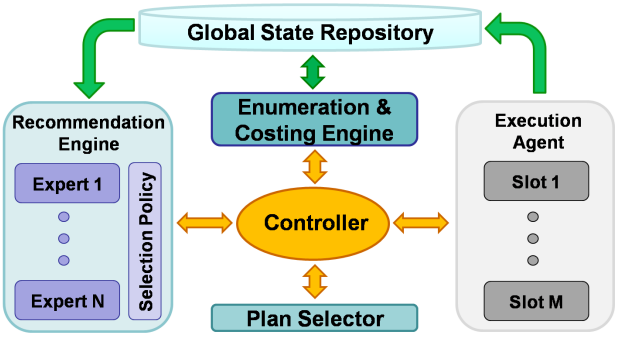


Figure 2: System architecture of Xplus

has no neighborhood to recommend, then the expert with the next highest priority is consulted. By default, priorities are assigned to experts in decreasing order of the degree of exploitation they do. Thus, Pure Exploiter has the highest priority, followed in order by Join Shuffler, Base Changer, and Pure Explorer. Overall, this strategy realizes a common (greedy) approach that humans use when faced with an exploitation versus exploration problem: explore only when further exploitation is not possible currently.

Reward-based Policy: This policy consults experts based on an online assignment of rewards that reflects how well the recommendations from each expert have performed in the past. Each time a new recommendation is needed, the expert with the current highest reward is consulted. If this expert has no neighborhood to recommend, then the expert with the next highest reward is consulted; and so on. Rewards are assigned as follows: If the overall least-cost plan changes based on an expert E's recommendation, then E's reward is incremented by 1; otherwise, it is reduced by 1.

3.3 Picking the Plan to Run in a Neighborhood

Once a neighborhood N is selected for coverage, the next step is to pick the least-cost (possibly-modified) plan p_{run} such that a run of p_{run} will bring AC values for all expressions in *MCS(N)*. *MCS(N)*, called the *missing cardinality set* of N, is the subset of expressions in *CS(N)* for which accurate cardinalities are currently unavailable. Xplus uses the following algorithm:

(a) Use intra transformations to enumerate all plans in N.
(b) For each plan $p \in N$, generate plan p' that has any modifications needed to collect all AC values for *MCS(N)*. Find the cost of p'.
(c) Pick p_{run} as the plan p' with least cost over all plans from (b).

Xplus supports two general plan-modification techniques for the nontrivial Step (b): subplan identification and additional scans.

Subplan identification: This technique finds the smallest connected subplan of p, starting from the operators at the leaves of of p, whose execution will bring all AC values for *MCS(N)*.

Additional scans: While most needed AC values correspond to the output cardinality of some operator in p, there are exceptions that need to be handled: (i) an indexed nested-loop join (INLJ) will not collect the inner table's cardinality, and (ii) table scans or index-based access in the presence of filters containing ANDs/ORs of individual predicates may not collect AC values for specific predicate combinations needed for costing [8]. Fortunately, both exceptions arise at leaf operators of p. Thus, Xplus addresses them by adding additional table or index scans to p in a cost-based manner.

The above plan-modification techniques were sufficient for PostgreSQL plans. (No changes were made to the PostgreSQL execution engine's source code.) Query plans in systems like IBM DB2 pose other exceptions like early exit from pipelines. Adding (blocking) materialization operators to plans for statistics collection is a plan-modification technique that can handle such exceptions [20].

4. IMPLEMENTATION OF XPLUS

4.1 Architecture

Xplus consists of six major components as shown in Figure 2:

- *Global State Repository*, which stores the data structures for neighborhoods and cardinalities described in Section 2.2, and the conventional database statistics from the system catalog.
- *Enumeration and Costing Engine*, which enumerates neighborhoods and plans as explained in Section 3.1, and estimates plan costs based on the information in the Repository.
- *Recommendation Engine*, which uses Experts and a Policy to recommend neighborhoods to cover as described in Section 3.2.
- *Plan Selector*, which selects the least-cost (sub)plan to collect the missing AC values in each recommended neighborhood, as described in Section 3.3.
- *Execution Agent*, which schedules selected (sub)plans for execution based on the specified resource constraints. Monitoring information from each execution is added to the Repository.
- *Controller*, which shepherds each input query through its lifecycle by invoking the above components appropriately. Section 4.2 describes multiple controllers implemented in Xplus.

We implemented the Xplus architecture following two guidelines:

1. It should be possible to implement Xplus with minimal changes to database internals, and without any major loss of efficiency. Requiring significant changes to database internals as a prerequisite would considerably reduce the chances of adoption.
2. If the SQL tuning feature of Xplus is not implemented directly by a database system's optimizer, then users should still be able to use Xplus as a tuning tool that coexists with the optimizer.

Accordingly, Xplus is implemented currently as a Java application that interacts with the database system through a well-defined interface provided by the system. SQL database systems contain external or internal interfaces to: (a) get cardinality and cost estimates for physical plans, and (b) run a specified plan and collect AC values for operators during the run. We implemented a new server command in PostgreSQL to expose its interface for costing and running plans to external applications [15]. Since the plan neighborhood abstraction, central to how Xplus works, is not present in current optimizers, we developed plan transformations as described in Section 3.1 and Appendix B.2. Overall, only minor changes were needed to PostgreSQL internals to support SQL tuning with Xplus. No changes were made to PostgreSQL's execution engine.

4.2 Extensibility Features

Xplus provides three dimensions for extensibility: adding new experts, new selection policies, and new controllers. SQL tuning problems that are hard to fix in a commercial database system usually get referred to the optimizer developers. Based on the reports seen over time, the developers may notice a defect in the optimizer that causes it to pick poor plans in certain conditions. Rather than modifying the optimizer and thoroughly testing the new version, an easy temporary fix can be to release a new Xplus expert that spots the mistake pattern and recommends plans to correct it. The expert is dropped once the optimizer is corrected and tested (which is very time consuming). This scenario illustrates one of the many positive impacts that Xplus can have on optimizer development and usage.

Adding a new expert, selection policy, or controller involves implementing specific interfaces defined by Xplus. We used this feature to implement five different controllers, described next. Recall that a controller is responsible for taking a given query through its entire lifecyle (tuning or conventional processing) in the system.

Plan-first-execute-next controller: This non-tuning controller enables Xplus to simulate the conventional query lifecyle: get the

least estimated cost plan in the plan space, and run it to generate the query results.

Serial (experts) controller: This controller repeatedly invokes the Xplus components in sequence until the stopping condition is met. The Recommendation Engine picks the next neighborhood N to cover in consultation with the Experts and the Selection Policy. N is given to the Plan Selector for selecting the least-cost plan to run to collect all missing AC values for *CS(N)*. The returned (sub)plan is run by the Execution Agent subject to the resource constraints specified. The new monitoring data is entered into the Repository.

Parallel (experts) controller: This controller runs the Recommendation Engine, Enumeration and Costing Engine, Plan Selector, and Execution Agent concurrently to enable inter-component parallelism in Xplus. The Parallel controller provides multiple benefits:

- If MPL_T (recall Section 2.1) is set greater than 1, then these many (sub)plans will be run concurrently.
- If new AC values from a plan execution are available when a costing cycle completes, another cycle is started to find the new least-cost plan; which helps when the plan space is large.
- Running the Recommendation Engine and Plan Selector in parallel with other components can hide plan recommendation latency in the presence of complex experts or plan modifications.

Leo controller: This controller implements tuning as done by the Leo optimizer [23]. In Xplus, the Leo controller effectively means using the serial controller with the Pure Exploiter as the only expert. Whenever the current plan finishes, the Pure Exploiter is consulted for the neighborhood to cover next. The Leo controller cannot make further progress when the Pure Exploiter gets livelocked.

ATO controller: This controller implements how Oracle's ATO [6] performs SQL tuning by collecting cardinality values for per-table filter predicates and two-way joins. After these cardinality values are collected, the new least-cost plan is recommended. Oracle's ATO estimates cardinality values using random sampling. Since PostgreSQL's execution engine has no external support for sampling, the ATO controller collects accurate AC values using the least-cost scan operator for filter predicates, and the least-cost join operator for two-way joins.

We could not implement controllers for other related work like [3, 5, 8, 20] because they all require nontrivial changes to the plan execution engine in the database. Such changes to the execution engine can help Xplus collect AC values faster and more efficiently.

4.3 Efficiency Features

Table 2 gives a summary of the features in Xplus that reduce the time to find a better plan as well as make Xplus scale to large queries. The first two features in Table 2 have been the most useful, so we will discuss them next. The other features are discussed in Appendix B.4. The first four features in Table 2 are fully integrated into Xplus, while the last three are left for future work.

Use of Parallelism: In addition to incorporating inter-component parallelism through the Parallel Controller, Xplus implements intra-component parallelism in multiple places. The Execution Agent can schedule multiple plans to run in parallel based on the MPL_T setting. The Costing Engine, which leverages the plan space partitioning imposed by neighborhoods, can cost plans in parallel. The Recommendation Engine can invoke different experts in parallel.

Executing Subplans instead of Full Plans: The Plan Selector implements this optimization as described in Section 3.3, giving major efficiency improvements as we will see in Section 5.

5. EXPERIMENTAL EVALUATION

The purpose of our evaluation of Xplus is threefold. First, we evaluate the effectiveness and efficiency of Xplus in tuning poorly-performing queries. Second, we compare Xplus with previous work. Finally, we evaluate different expert-selection policies, combinations of experts, and the impact of the efficiency features. All experiments were run on an Ubuntu Linux 9.04 machine, with an Intel Core Duo 3.16GHz CPU, 8GB of RAM, and an 80GB 7200 RPM SATA-300 hard drive. The database server used was PostgreSQL 8.3.4. We used the TPC-H Benchmark with a scale factor of 10. We used an index advisor[3] to recommend indexes for the TPC-H workload. All table and column statistics are up to date except when creating problem queries due to stale statistics. Unless otherwise noted, all results were obtained using the Parallel Experts Controller with MPL_T=2 and the Priority policy with all experts.

Feature	Description
Use of parallelism	Leverage intra- and inter-component parallelism
Plan modification	Run lower-cost subplans instead of full plans
Prioritize neighborhoods	Avoid neighborhoods with bad subplans that cannot lead to a plan better than the current best plan
Table-level preprocessing	Perform optimized table/index scans to collect table-level AC values in a fast preprocessing step
Materialization	Store and reuse intermediate results during tuning
Sampling	Run (sub)plans on samples instead of the full data
Modifications to execution engine	Make modifications to the execution engine as in [5, 8, 20] to increase monitoring capabilities of plans

Table 2: Summary of features that improve efficiency of Xplus

	TPC-H Queries								
Tuning Scenario Class	2	5	7	9	10	11	16	20	21
Query-level issues	✓		✓	✓	✓			✓	
Data-level issues	✓	✓	✓			✓			✓
Statistics-level issues		✓			✓		✓		✓
Physical-design issues			✓	✓			✓	✓	

Table 3: Tuning scenarios created with TPC-H queries

5.1 Tuning Scenarios

We will present the evaluation of Xplus in terms of *tuning scenarios*, where a query performs poorly due to some root cause. Four classes of query tuning scenarios are common in practice:

1. *Query-level issues*: A query may contain a complex predicate (e.g., with a UDF) for which cardinality estimation is hard.
2. *Data-level issues*: Real-life datasets contain skew and correlations that are hard to capture using common database statistics.
3. *Statistics-level issues*: Statistics may be stale or missing.
4. *Physical-design issues*: The optimizer may not pick a useful index, or it may pick an index that causes a lot of random I/O.

We created multiple instances per tuning scenario class. Query-level issues were created by making minor changes to the TPC-H query templates, mainly in the form of adding filter predicates. Data-level issues were created by injecting Zipfian skew into some columns. We decreased the amount of statistics collected by PostgreSQL for some columns to create issues at the statistics and physical design levels. Table 3 summarizes the issues that were caused for each TPC-H query. Often problems are due to some combination of multiple root causes, which is reflected in Table 3.

5.2 Overall Performance of Xplus

Table 4 provides the results for nine different tuning scenarios. (All plan running times shown are averaged over six runs.) In all nine cases, Xplus found a better execution plan, offering an average speedup of 7.7 times faster compared to the original plan (selected by the PostgreSQL query optimizer) to be tuned. In three cases, Xplus found a new plan that is over an order of magnitude faster.

The last two columns of Table 4 show the time Xplus takes to find the better plan. The absolute times (second-last column) are

[3]The DB2 index advisor (db2advis) was used since we have observed that its index recommendations work well for PostgreSQL.

Query	Run Time of Original PostgreSQL Plan (sec)	Run Time of Xplus Plan (sec)	Speedup Factor	Number of Subplans Xplus Ran	Time to Find Xplus Plan	
					Absolute (sec)	Normalized (Col6/Col2)
2	8.67	0.59	14.8	5	40.42	4.66
5	1037.80	399.01	2.6	8	149.76	0.14
7	257.55	21.38	12.0	6	131.58	0.51
9	1722.27	754.82	2.3	8	870.78	0.51
10	2248.52	695.70	3.2	4	149.15	0.07
11	20.00	3.55	5.6	2	29.11	1.46
16	15.90	0.77	20.7	2	27.04	1.70
20	3.36	2.32	1.4	4	7.13	2.13
21	509.51	72.17	7.1	4	45.83	0.09

Table 4: Overall tuning results of Xplus for TPC-H queries

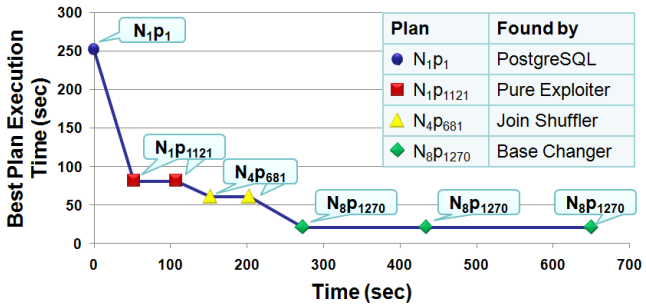


Figure 3: Progress of the execution time of the best plan in the covered space as Xplus tunes TPC-H Query 7. Serial Experts Controller with Priority policy and all four experts is used

small, which shows the high degree of efficiency that our implementation achieves. In particular, the last column shows *normalized tuning time*, which is the ratio of the time taken by Xplus to find the better plan to the running time of the original plan to be tuned. The low values in this column clearly demonstrate how Xplus gives its benefits in the time it takes to run the original plan a very small number of times (often < 2).

Figure 3 shows the execution timeline of Xplus while tuning TPC-H Query 7. The y-axis shows the execution time of the best plan found so far in the covered space. The plan found by the PostgreSQL optimizer is N_1p_1 (plan p_1 in neighborhood N_1) which runs in 257.55 seconds. Let us see the role of the experts in Xplus in this tuning task. For ease of plotting the timeline, the Serial Controller with the Priority policy and all four experts was used. First, a neighborhood recommended by the Pure Exploiter led to the discovery of plan N_1p_{1121}, which gave a speedup factor of 3.1. The Pure Exploiter livelocked at this point. The Join Shuffler then recommended a neighborhood that led to plan N_4p_{681}; increasing the speedup to 4.1. It took a recommendation from the exploration-heavy Base Changer for Xplus to find plan N_8p_{1270} with a speedup of 12. All neighborhoods were covered by the execution of 7 subplans (not full plans). Recall the strong guarantee that Xplus provides once all neighborhoods are covered (Property 2).

This tuning task is an excellent example of how exploitation and exploration are both needed to reach the optimal plan. Appendix C gives a more detailed description that also discusses how, unlike Xplus, the Leo and ATO controllers failed to find the optimal plan.

5.3 Comparison with Other Approaches

We now compare Xplus with two other SQL tuning approaches: Leo and Oracle's ATO using the respective controllers discussed in Section 4.2. For the same nine tuning scenarios from Table 4, Figure 4 shows the speedup factor of the plans produced by the three approaches compared to the original plan to be tuned. Xplus found a better plan than Leo in 4 cases, offering up to an order of magnitude additional speedup. Xplus found a better plan than ATO in 7

Query	Xplus		Leo Controller		ATO Controller	
	Time	Result	Time	Result	Time	Result
2	4.66	Success	5.09	Failure(2.9)	4.99	Failure(2.2)
5	2.56	Guarantee(2.6)	0.57	Failure(2.6)	1.92	Failure(1.5)
7	0.51	Success	0.26	Failure(3.2)	1.03	Failure(4.2)
9	2.91	Guarantee(2.3)	0.91	Failure(1.4)	2.13	Failure(1.5)
10	0.07	Guarantee(3.2)	0.03	Failure(1.9)	0.23	Failure(3.2)
11	1.46	Success	0.14	Success	0.54	Success
16	1.70	Success	0.10	Success	1.35	Failure(2.8)
20	2.23	Guarantee(1.4)	2.12	Failure(1.4)	3.07	Failure(1.0)
21	0.09	Success	0.01	Success	0.59	Failure(1.9)

Table 5: Tuning results of Xplus, Leo controller, and ATO controller when asked to find a 5x better plan. Time is normalized over the execution time of the original PostgreSQL plan

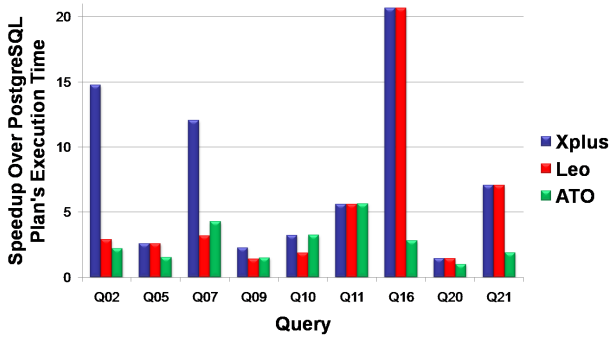


Figure 4: Speedup from Xplus, Leo, and ATO controllers

cases, with similar improvements. The performance advantages of Xplus are more prominent for more complex queries.

Motivated by the SLO setting from Section 1, Table 5 shows the performance of Xplus, Leo, and ATO controllers for the following tuning task per query Q: *find a plan that is 5x faster than the current plan picked by the PostgreSQL optimizer for* Q. For each approach, we show its normalized tuning time and *result*. For the Leo and ATO controllers, the result is one of: (i) *Success*, if a plan with $\geq$5 speedup is found; or (ii) *Failure*(α), if the controller could only find a plan with α<5 speedup. In contrast, when Xplus fails to find a plan with $\geq$5 speedup, it provides the guarantee *Guarantee*(α): for the given database configuration and optimizer cost model, the optimal plan for Q only gives α speedup. With this knowledge, the user or tuning tool can plan for the disruptive changes needed to the physical design, server parameters, or resource provisioning to get the desired performance for the query.

Xplus finds a 5x faster plan in five cases in Table 5, and provides a strong guarantee in the rest. The Leo and ATO controllers succeed in only three cases and one case respectively. The Leo controller fails to complete a task because it runs into a livelock, whereas the ATO controller fails because the cardinality estimates gathered from sampling tables and two-way joins are not enough to produce a plan with the desired performance.

5.4 Internal Comparisons for Xplus

Figures 5 and 6 illustrate an important trend that emerged in our evaluation. These figures consider five strategies for plan recommendation: Priority, Round Robin, and Rewards, each with all four experts; and Priority with (a) the Pure Exploiter and Pure Explorer (called *Exploiter-Explorer*), and (b) the Pure Explorer only (called *Explorer-Only*). These strategies are compared based on *convergence time* (how soon they found the best plan), as well as the *completion time* (how long they took to cover all neighborhoods).

Note from Figures 5 and 6 that exploration-heavy policies (like Explorer-Only and Exploiter-Explorer) take longer to converge, but lead to lower completion times. Exploitation targets missing statistics related to the current least-cost plan, which leads to better con-

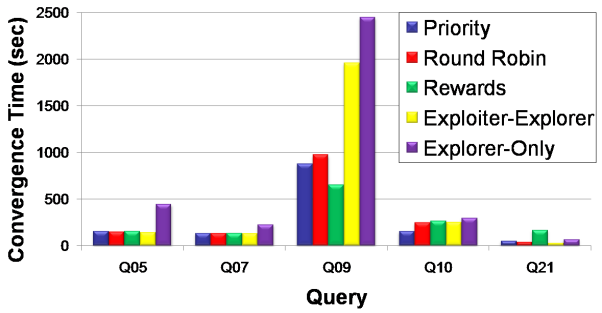


Figure 5: Convergence times for the expert-selection policies

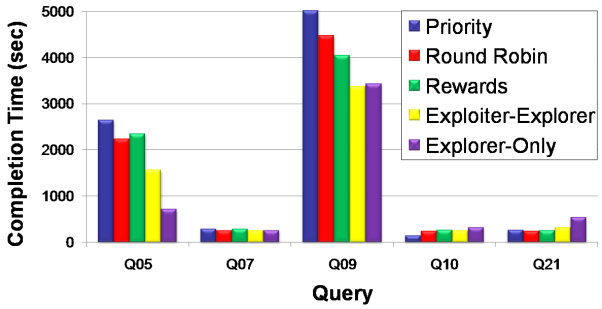


Figure 6: Completion times for the expert-selection policies

vergence. However, the time to gather all statistics is longer as exploitation makes small steps towards this goal. Exploration brings in more information in each step, often decreasing the total number of executed (sub)plans and the overall completion time.

Based on these observations, we offer the following guideline to choose the policy and experts for a SQL tuning task. If the user or DBA wishes to find the best plan possible (e.g., to decide whether disruptive tuning can be avoided), then she should select an exploration-heavy strategy. On the other hand, if she is interested in quick improvements to the current plan, then a strategy that favors exploitation over exploration is more suitable.

Figure 7 shows the impact of the two important efficiency features of Xplus: use of parallelism and running subplans instead of full plans whenever possible. Use of subplans is particularly beneficial for complex and long-running queries. For example, Xplus ran 8 subplans to cover all neighborhoods for TPC-H Query 5. Most of these subplans contained only around half of the tables in a full plan for the query, causing Xplus to complete four times faster.

6. DISCUSSION

The thesis of this paper is that the query optimizer can automate the important task of SQL tuning, and is the right entity to do so. To support this thesis, we designed, implemented, and evaluated a novel query optimizer called Xplus. An Xplus user can mark a repeatedly-run query for which she is not satisfied with the performance of the plan being picked; and Xplus will try to find a new plan that gives the desired performance. Xplus differs from regular query optimizers in its ability to run plans proactively, and to collect monitoring data from these runs to diagnose its mistakes as well as to identify better plans. We made the following contributions:

- We introduced the abstraction of plan neighborhoods in the physical plan space. This abstraction is a key contributor to the effectiveness and efficiency of Xplus.
- We showed how the two conflicting objectives of exploitation and exploration need to be balanced in effective SQL tuning. Xplus uses an architecture based on multiple SQL-tuning experts and an arbitration policy to achieve this balance.
- We validated the promise of Xplus through an extensive evaluation based on tuning scenarios that arise in practice.

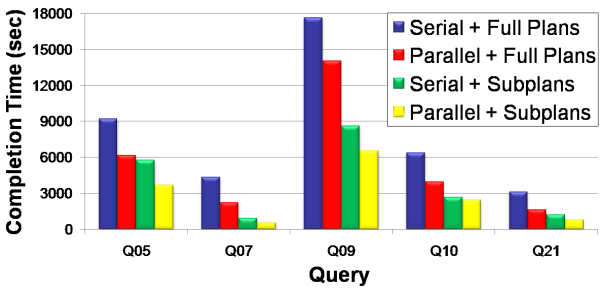


Figure 7: Impact of the efficiency features in Xplus

7. REFERENCES

[1] M. Andrei and P. Valduriez. User-Optimizer Communication using Abstract Plans in Sybase ASE. In *Proc. of VLDB '01*. ACM, 2001.

[2] G. Antoshenkov and M. Ziauddin. Query Processing and Optimization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of SIGMOD '00*. ACM, 2000.

[4] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proc. of SIGMOD '05*, 2005.

[5] S. Babu, P. Bizarro, and D. DeWitt. Proactive Reoptimization. In *Proc. of SIGMOD '05*. ACM, 2005.

[6] P. Belknap, B. Dageville, K. Dias, and K. Yagoub. Self-Tuning for SQL Performance in Oracle Database 11g. *Intl. Conf. on Data Engineering*, 2009.

[7] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power Hints for Query Optimization. In *Intl. Conf. on Data Engineering*, 2009.

[8] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A Pay-As-You-Go Framework for Query Execution Feedback. In *Proc. of VLDB '08*. VLDB Endowment, 2008.

[9] C. M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation using Query Feedback. *SIGMOD Record*, 23(2):161–172, 1994.

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

[11] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. In *Proc. of VLDB '09*, 2009.

[12] J. C. Gittins and D. M. Jones. A Dynamic Allocation Index for the Sequential Design of Experiments. *Progress in statistics (European Meeting of Statisticians)*, 1972.

[13] G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. In *Proc. of SIGMOD '87*. ACM, 1987.

[14] P. J. Haas, I. F. Ilyas, G. M. Lohman, and V. Markl. Discovering and Exploiting Statistical Properties for Query Optimization in Relational Databases: A Survey. *Statistical Analysis and Data Mining*, 2009.

[15] H. Herodotou and S. Babu. Automated SQL Tuning through Trial and (Sometimes) Error. In *DBTest '09: Proc. of the 2nd Intl Workshop on Testing Database Systems*. ACM, 2009.

[16] IBM Corp. *DB2 Information Center*. http://publib.boulder.ibm.com/infocenter/db2luw/v8.

[17] IBM DB2. *Giving optimization hints to DB2*, 2003. http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db2.admin/ p91i375.htm.

[18] A. Kemper, G. Moerkotte, and K. Peithner. A Blackboard Architecture for Query Optimization in Object Bases. In *Proc. of VLDB '93*. ACM, 1993.

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

[20] V. Markl, V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. Robust Query Processing through Progressive Optimization. In *Proc. of SIGMOD '04*. ACM, June 2004.

[21] F. Olken and D. Rotem. Random Sampling from Databases: A Survey. *Statistics and Computing*, 5:25–42, 1995.

[22] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of SIGMOD '79*. ACM, 1979.

[23] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's Learning Optimizer. In *Proc. of VLDB '01*. Morgan Kaufmann Publishers Inc., 2001.

APPENDIX

A. OTHER APPROACHES TO SQL TUNING

In this section, we discuss current approaches to SQL tuning and provide a detailed comparison with Xplus. Table 6 provides a high-level summary of the similarities and differences between Xplus and other approaches with respect to various important properties.

Using Feedback from Query Execution: Leo [23] corrects cardinality estimation errors made by the query optimizer by comparing EC values with AC values obtained when plans run [9]. This approach can find a better plan for a poorly-performing query Q over multiple executions of Q or of queries with similar subexpressions. The pay-as-you-go approach took this idea further using proactive plan modification and monitoring techniques to measure approximate cardinality values for subexpressions, in addition to the subexpressions contained in a running plan [8]. While query execution feedback is related closely to how Xplus performs SQL tuning, there are some key differences between Xplus and [8, 23]:

- SQL tuning is inherently an unpredictable and risky process in that a plan better than the optimizer's original pick p may be found only after some plans worse than p are tried. Given how difficult query optimization is, there is always an element of trial-and-error in SQL tuning. Furthermore, experiences with Leo show that incorporating some AC values alongside EC values can cause optimizers to pick plans whose performance is worse than before [19]. Thus, attempting SQL tuning directly on queries being run by users runs the risk of unpredictable behavior and performance regression. DBAs and users usually prefer predictable, possibly lower, performance over unpredictable behavior [4]. For this reason, unlike [8, 23], Xplus deliberately keeps the SQL tuning path separate from the normal path of submitted queries.
- The concept of balancing the exploitation and exploration objectives explicitly in SQL tuning is unique to Xplus. Leo and pay-as-you-go are exploitation-heavy approaches, ignoring the uncertainty in estimates when picking the plan for the query.
- A serious concern with using an exploitation-heavy approach is the possibility of a livelock (see Section 3.2) because the subexpressions produced by a plan dictate which set of actual cardinality values are available from running that plan.
- Unlike Leo and Xplus, implementing the pay-as-you-go approach (and related ones like [5, 20]) in a database system requires nontrivial changes to the plan execution engine.
- Execution of plans brings in valuable information like EC, AC, and EAC cardinality values. Other types of information that Xplus can track based on plan execution include estimated and actual costs (including *estimated-actual costs* similar to EAC), I/O patterns, and resource bottlenecks. Significant differences between actual costs and the corresponding estimated-actual costs may indicate errors in the optimizer cost model.

Oracle's Automatic Tuning Optimizer (ATO): Xplus shares common goals with Oracle's ATO [6], but differs in the algorithms and system architecture. While Leo and pay-as-you-go focus on exploitation, ATO is on the exploration side. When invoked for a query Q, ATO does additional processing to reduce the uncertainty in cardinality estimates for Q. First, ATO collects random samples from the base tables to validate the cardinality estimates of Q's filter predicates. Given more time, ATO performs random sampling to validate all two-way join cardinality estimates (possibly, up to all n-way joins). ATO uses a sandboxed environment for the additional processing to limit the overhead of the tuning process on the rest of the database workload. If the new estimates cause the least-cost plan to change, then ATO compares the performance of the new plan against the old one by running both plans in a competitive fashion. Unlike Xplus, ATO has features to recommend new statistics to collect, indexes to create, and rewrites to the query text.

Property	Xplus	Leo	Pay-As-You-Go	ATO
Balanced use of exploitation and exploration	Yes	No	No	No
Support for SLO tuning tasks	Yes	No	No	No
Risk of unpredictable changes to user-facing query behavior	No	Yes	Yes	No
Requires changes to the query execution engine	No	No	Yes	No
Provides optimality guarantees for given database configuration and optimizer cost model	Yes (Prop. 2, Sec. 2.3)	No	No	No
Use of parallelism	Yes	No	No	Possible
Use of collected statistics to improve plans for other queries	Possible	Yes	Yes	Possible
Potential to address errors in the optimizer cost model	Yes	No	No	Yes
Possibility for running into a livelock in the SQL tuning process (Section 3.2)	Depends on choice of experts	Yes	Yes	No
Use in fully automated tuning	Possible	Yes	Yes	Yes

Table 6: Comparison of Xplus, Leo, Pay-As-You-Go, and ATO

Adaptive Query Processing: Xplus is one point in the design spectrum that includes a long line of work on adaptive query processing [10]. The Rdb system introduced *competition-based* query plan selection, namely, running multiple plans for the same query concurrently, and retaining the best one [2]. Database systems for emerging application domains (e.g., MongoDB) are using this concept to address the lack of statistics in these settings.

Eddies [3] identified the relationship of adaptive query processing to *multi-armed bandit (MAB)* problems from machine learning [12]. The study of MAB problems has led to theory and algorithms to balance the exploitation-exploration tradeoff under certain assumptions, including algorithms to control multiple competing experts [12]. Xplus uses a similar approach by designing experts for SQL tuning who recommend new plans to try based on the information available so far. While the experts in Xplus are static, small in number, and recommend query plans, each expert in Eddies and Rdb is a candidate plan for the query; which makes the architecture and algorithms of Xplus very different from that of [2, 3]. Experts have also been used in query optimizers to exercise rewrite rules and heuristics during the optimization of a query [18].

Multiple optimization levels: Most current optimizers provide multiple levels of optimization. For example, when a query is optimized in IBM DB2's most powerful 9th level, all available statistics, query rewrite rules, access methods, and join orders are considered [16]. The design of *X*plus (which stands for 10+) came from considering what hypothetical classes 10 and higher of DB2's optimizer could usefully do. Our answer is that these higher levels will execute selected (sub)plans proactively, and iterate based on the observed information; a challenging task left to DBAs today.

B. FURTHER IMPLEMENTATION DETAILS

We present details on the Xplus implementation that could not be presented in the main body of the paper due to space constraints.

B.1 Generation of Cardinality Sets

Given an operator instance O in a plan p, the *Generate_CS(O,p)* function will return *CS(O)*, the cardinality set of O, which consists of the relational algebra expressions whose cardinalities are needed to find O's execution cost in p. Table 7 lists the cardinality sets returned by the *Generate_CS* function for the physical opera-

Physical Operator in PostgreSQL	Cardinality Set
Sequential Table Scan (TS) on table S	S
Index Scan (IS) on table S with index predicate p	$\sigma_p(S)$
Bitmap Index Scan (BIS) on table S with index predicate p (produces a bitmap of rows matching p)	S, $\sigma_p(S)$
Bitmap AND/OR Operator over two or more Bitmap Index Scans (produces a bitmap representing the ANDing/ORing of the input bitmaps)	$\emptyset$[5]
Bitmap Heap Scan over expression L (fetches rows based on input bitmap from a Bitmap Index Scan or Bitmap AND/OR Operator)	L
Hash Join (HJ) over child expressions L and R	L, R, $L \bowtie R$
Merge Join (MJ) over child expressions L and R	L, R, $L \bowtie R$
Nested Loop Join (NLJ) over child expressions L and R	L, R, $L \bowtie R$
Index Nested Loop Join (INLJ) over child expressions L and R	L, $L \bowtie R$
Plain Aggregate (no grouping) over expression L	L
Hash/Sort-based GroupBy/Aggregate over expression L with grouping attribute a and aggregate function f	L, ${}_a\mathcal{G}_f(L)$[6]
Unique Operator over expression L and attribute a	L, ${}_a\mathcal{G}(L)$[7]
Sort Operator over expression L	L
Materialize Operator over expression L	L

Table 7: Cardinality sets returned by the Generate_CS function for each physical operator in PostgreSQL

tors in PostgreSQL. The expressions in *CS(O)* are a subset of the relational algebra expressions that define each of O's inputs and outputs. For example, for a hash join operator H over two subplans representing the relational algebra expressions L and R in a plan p, *Generate_CS(H,p)* will return the cardinality set $\{L, R, L \bowtie R\}$. According to PostgreSQL's cost model for a hash join operator, the cardinalities of L, R and $L \bowtie R$[4] are needed to cost H.

B.2 Transformations

Multiple components of Xplus, including the Plan Selector and the experts in the Recommendation Engine, need to enumerate and reason about plan neighborhoods and physical plans. As discussed in Section 3.1, plan transformations serve this purpose. Each transformation τ in Xplus is implemented as a function that operates on a data structure representing a physical plan p for a query Q. If τ is applicable to p, then the function's output is a data structure representing a plan p' ($p' \neq p$) for Q. If τ is an intra transformation, then p' will belong to the same neighborhood as p. If τ is an inter transformation, then p' will be in a different neighborhood from p. Applying these transformations multiple times enables Xplus to enumerate all of Q's plans and neighborhoods.

Like many query optimizers (including PostgreSQL), Xplus uses non-cost-based query rewrite rules to identify the *select-project-join-aggregation (SPJA)* blocks in the query. Each SPJA block is optimized separately to produce a plan per block; and these plans are connected together to form the execution plan for the full query. Intra and inter transformations in Xplus are applied to plans for SPJA blocks. Recall from Section 2.1 that a tuning session of Xplus takes the input query Q and its current (unsatisfactory) plan p as input. Xplus works with the SPJA blocks in p.

Intra Transformations: Intra transformations are applied to a single operator in a plan to generate a different plan in the same neighborhood. These transformations belong to one of two classes:

1. *Method Transformations*, which change one operator implementation (method) to another implementation of the same underlying logical operation. Instances of this class include: (i) Transforming a scan method to a different one (e.g., transforming a full table scan on a table to an index scan); (ii) Transforming a join method to a different one (e.g., transforming a hash join to a merge join); (iii) Transforming a grouping and aggregation method to a different one (e.g., transforming a sort-based aggregation operator to a hash-based one).
2. *Commutativity Transformations*, which swap the order of the outer and inner subplans of a commutative operator. For example, transforming $L \bowtie R$ to $R \bowtie L$, for subplans L and R.

It is important to note that some transformations may not be possible on a given plan. For example, a table scan on table R cannot be transformed into an index scan unless an index exists on R. Also, additional changes may be required along with the application of a transformation in order to preserve plan correctness. For instance, a merge join requires both of its subplans to produce sorted output. If they do not produce sorted output, then a transformation from a hash join to a merge join must add sort operators above the subplans. Note that the addition of these sort operators will not change the cardinality set of the plan (recall plans p_1 and p_2 in Figure 1(a)).

Inter Transformations: When applied to a plan p, an inter transformation produces a plan p' in a different neighborhood. That is, *CS(p)* $\neq$ *CS(p')*. Inter transformations predominantly apply across multiple operators in a plan. (It is theoretically possible that changing the implementation method of an operator in a plan produces a plan in a different neighborhood.) The main inter transformation swaps two tables that do not belong to the same join in a join tree. As with intra transformations, any changes required to preserve the correctness of the new plan are done along with the inter transformation. The two other inter transformations are: (i) Pulling a filter operation F over an operator (by default, F is done at the earliest operator where F's inputs are available in the plan); (ii) Pushing a grouping and aggregation operator below another operator (by default, the grouping and aggregation operator is done after all filters and joins). Applying these transformations multiple times, allows Xplus to generate plans from all neighborhoods.

Our running example star-join query has six neighborhoods, each with its own distinct join tree (see Figure 1(a)). Plan p_2 can be produced from p_1 by applying two intra transformations: a method transformation that changes the middle join from a hash join to a merge join, and a commutativity transformation that changes $R \bowtie S$ to $S \bowtie R$. Plan p_3 can be produced from p_1 by applying one inter transformation (swap T and S) and one intra transformation (change $R \bowtie T$ to $T \bowtie R$).

B.3 Experts

This section describes the implementation of the experts (introduced in Section 3.2.1) used by the Recommendation Engine.

Pure Exploiter Expert: Since exploitation is the sole objective of the Pure Exploiter, it simply recommends the neighborhood N_{opt} containing the plan with the lowest estimated cost in the plan space, based on the currently available information. The least-cost plan and its neighborhood are readily available in the Repository. N_{opt} is recommended if it has not been covered yet; otherwise, the expert has run into a livelock and no neighborhood is recommended.

Each time new AC values are entered for a set of expressions E into the Cardinality Table in the Repository, the Enumeration and Costing Engine finds all (uncovered) neighborhoods N for which *CS(N)*$\cap E \neq \emptyset$. The least-cost plans in these neighborhoods may have changed, so they are recomputed and the Neighborhood Table is updated as needed. Thus, the Cardinality and Neighborhood Ta-

[4] $|L \bowtie R|$ is used for estimating the CPU cost of the hash join.

[5] PostgreSQL costs a Bitmap AND/OR Operator by adding a fixed cost to the cost of the child BISs. No cardinality values are used.

[6] ${}_a\mathcal{G}_f(L)$ represents the relational algebra expression for the grouping of tuples from L on attribute a and the application of the aggregate function f on the groups.

[7] ${}_a\mathcal{G}(L)$ is similar to ${}_a\mathcal{G}_f(L)$ but without the aggregation. The groups represent the unique values.

bles make it efficient to incrementally maintain the least-cost plan by comparing the least-cost plan from each neighborhood (both covered and uncovered). For uncovered neighborhoods, the plan cost estimation is based on the available AC values and the EC values computed using statistics supplemented, as is common, with assumptions of uniformity and independence as well as the use of magic numbers. Uncertainty in these estimates is not taken into consideration while computing plan costs. Therefore, it is possible for the Pure Exploiter to recommend a bad neighborhood by mistake, just like a regular optimizer could select a bad plan.

Join Shuffler Expert: The Join Shuffler works as follows when invoked to get a neighborhood recommendation. It first gets the best plan p_{best} among all covered neighborhoods from the Repository. Since p_{best} belongs to a covered neighborhood, AC and EAC values are available for all operators in p_{best}; which is a required property as we will see shortly. The Join Shuffler then identifies the join in p_{best} with the highest *overestimated* join selectivity, and tries to push this join as low in the plan as possible (with any transformations required to maintain correctness as discussed in Appendix B.2). If the join can be pushed down and that leads to a new plan p_{new} in an uncovered neighborhood N_{new}, then N_{new} will be the neighborhood recommended by the Join Shuffler.

The degree of overestimation d in the selectivity of a join operator O is computed as the relative difference between O's EAC and AC values. That is, $d = \frac{EAC-AC}{AC}$. Large d for O is an indication that O's join selectivity is much smaller than what the optimizer estimated based on the current information in the Repository. Hence, pushing such joins down the join tree can potentially reduce the data flow up the tree, and decrease the overall cost of the plan.

If pushing down the join with the highest overestimated join selectivity does not lead to a plan in an uncovered neighborhood, then the same exercise is repeated for the join with the second highest overestimated join selectivity. This process is repeated until a plan in an uncovered neighborhood is found. If no such plan can be found, then the Join Shuffler is itself livelocked.

Base Changer Expert: When invoked for a recommendation, the Base Changer first gets the current least-cost plan p_{opt} in the entire space from the Repository. If p_{opt} is different from the last time the Base Changer was invoked, inter transformations are applied to p_{opt} to generate a set of plans that contains one plan per neighborhood. The plans in this set are then partitioned such that each partition contains all plans having the same *base*. The base of a plan p is defined as the first join to be executed in p. For example, the base of the plan p_1 in Figure 1(a) is $R \bowtie S$. Note that there can be multiple plans for each base. All plans belonging to the same partition are added to a queue in order of their generation through inter transformations starting from p_{opt}; which roughly orders the plans in each queue in decreasing order of similarity to p_{opt}.

Each queue is assigned to its corresponding base in an array of all bases. The Base Changer maintains this array of all possible bases for the query, and proceeds through it in a round-robin fashion. The base in the plan that was recommended last is kept track of so that the next recommendation can start from the next base in the array. Thus, fairness of exploration across all bases is enforced. When a base b is considered during the round-robin traversal of the array, the plan p_{hb} is removed from the head of b's queue. If p_{hb} belongs to an uncovered neighborhood Np_{hb}, then the Base Changer will recommend Np_{hb} as the neighborhood to be covered next. If p_{hb} belongs to a covered neighborhood, then the Base Changer proceeds to the next base in the array; p_{hb} is not considered again.

Assuming all neighborhoods have not been covered yet, the Base Changer will always have a neighborhood to recommend. That is, the Base Changer will not run into a livelock like the Pure Exploiter or the Join Shuffler. This property holds because each neighborhood is represented in the array, and the round-robin traversal ensures that each neighborhood will be considered eventually.

Pure Explorer Expert: For each uncovered neighborhood N_u in the space (found in the Neighborhood Table in the Repository), the Pure Explorer computes the number of expressions in the cardinality set of N_u whose AC values are not available in the Repository. The Pure Explorer will then recommend the neighborhood with the highest number of such expressions. Ties are broken arbitrarily. Like the Base Changer, the Pure Explorer will never livelock.

For example, suppose that neighborhood N_1 from our running example in Figure 1(a) has been covered. The Cardinality Table in the Repository contains AC values for each expression in the cardinality set of N_1 (seen in Figure 1(b)). Suppose the Pure Explorer has two options for the neighborhood to recommend: N_2 and N_6. In this case, the Pure Explorer will recommend covering N_6 since it will bring in AC values for two uncertain expressions, namely, $\sigma_p(R) \bowtie U$ and $\sigma_p(R) \bowtie T \bowtie U$; whereas covering N_2 will only bring in the AC value for $\sigma_p(R) \bowtie T$.

B.4 Other Efficiency Features

This section discusses the last five features from Table 2 that improve the efficiency of Xplus.

Prioritize neighborhoods: It is possible that some neighborhoods consist almost exclusively of highly suboptimal plans. Xplus proactively identifies and avoids such neighborhoods, even when not all AC values are available for them. As an example, consider plan p_4 belonging to neighborhood N_6 in Figure 1(a). Let AC values be available for $\sigma_p(R)$, U, and $\sigma_p(R) \bowtie U$, but not for the rest of the plan. Also suppose that the three available AC values are very high. Then, irrespective of what values the unknown cardinalities take, Xplus may discover that the cost of doing $\sigma_p(R) \bowtie U$ makes all plans in neighborhood N_6 worse than the overall best plan among the neighborhoods covered so far. If so, Xplus can avoid N_6.

Optimized preprocessing: Recall how the Plan Selector may need to add additional scans to plans to collect needed AC values. Xplus proactively identifies such exceptions during neighborhood enumeration, and does cost-based table/index scans per table to collect needed AC values in a preprocessing step. Table-by-table preprocessing is efficient because it makes better use of the buffer cache.

Use of materialization: Plans from different neighborhoods may share common subexpressions (e.g., $R \bowtie S$ for neighborhoods N_1 and N_4 in Figure 1(a)). Finding commonalities and creating materialized views help avoid recomputation during SQL tuning.

Use of sampling: Database systems have made giant strides in internal and external support for sampling [21]. Xplus could use sampling instead of executing (sub)plans on the full data.

Execution engine modifications: Xplus (especially the Plan Selector) can benefit from a number of plan-modification techniques proposed in the database research literature to increase statistics and cost monitoring capabilities during plan execution [5, 8, 20].

C. ZOOMING INTO TUNING OF QUERY 7

This section drills down into how Xplus, Leo, and ATO tune Query 7. The Xplus Serial Controller, with Priority policy and all four experts, is used to demonstrate the tuning process clearly.

C.1 Tuning Process of Xplus for Query 7

Figure 8 shows several execution plans for Query 7 that came up in the tuning process (recall Figure 3). All plans have a sort operator and an aggregate at the top which are omitted from the figure for clarity. Plan N_1p_1 (which denotes plan p_1 in neighborhood

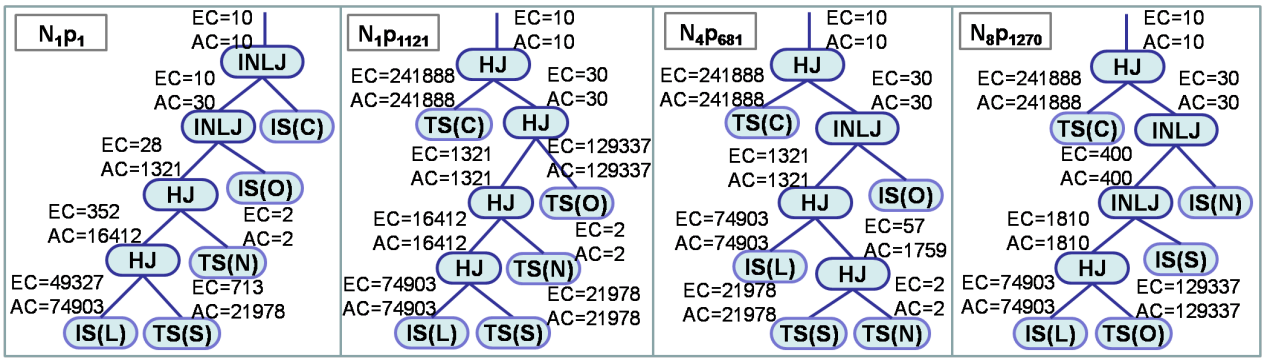


Figure 8: Execution plans for Query 7 (with one instance of the Nation table removed for clarity). N_1p_1 was chosen by the PostgreSQL optimizer, N_8p_{1270} by Xplus (Parallel Controller, Priority Policy, and all four Experts), N_1p_{1121} by the Leo controller, and N_4p_{681} by the ATO controller. We present the EC and AC values at the time each plan was suggested to be the best plan by Xplus

N_1) represents the plan selected and run by the PostgreSQL query optimizer. The poor performance of this plan causes Xplus to be invoked for tuning. Note the large cardinality estimation error for S, which propagates all the way up. Figure 8 shows the AC values available from the run of N_1p_1, which covers neighborhood N_1.

When Xplus updates plan costs based on the new AC values, N_1p_{1121} emerges as the least-cost plan in the covered space (only N_1 is covered currently). Correcting the underestimation caused the index nested loop joins in N_1p_1 to be replaced with hash joins over sequential scans in N_1p_{1121}. The new least-cost plan in the entire space is N_3p_{1121} (not shown in Figure 8) which has a different join order from N_1p_1 (tables O and N are swapped), but retains the index nested loop join over O. We ran plans N_1p_{1121} and N_3p_{1121} separately (as we did for all plans involved) to validate their performance. N_1p_{1121} was indeed better that the original N_1p_1, running in 80.8 seconds versus 251.7 seconds. However, plan N_3p_{1121} executed in 172.4 seconds, illustrating how correcting some estimation errors need not necessarily lead to a better plan.

Since the least-cost plan is N_3p_{1121} and neighborhood N_3 is uncovered, the Pure Exploiter will recommend N_3 next. A subplan gets run. The new AC values lead to the coverage of N_3, and the convergence of the least-cost plans in the covered and uncovered spaces to N_1p_{1121}. Now the Pure Exploiter is livelocked, so the Priority Policy consults the Join Shuffler. After executing a subplan resulting from the Join Shuffler's recommendation, N_4p_{681} becomes the least-cost plan in the covered space. The different join order (with new base $S \bowtie N$) leads to a performance improvement.

After recommending one more neighborhood, the Join Shuffler also runs into a livelock. The next recommendation comes from the Base Changer which leads Xplus to plan N_8p_{1270} (with base $L \bowtie O$) as the best plan in the covered space; and eventually the entire space. Specifically, after covering a neighborhood recommended by the Base Changer, Xplus was able to spot that $L \bowtie O$ (whose join selectivity was severely overestimated) has few joining tuples.

Overall, Xplus took 11 minutes to cover the entire plan space which contains 14 neighborhoods and 22,400 plans. Xplus only ran 7 subplans. 2 of the 11 minutes were spent in plan costing because of the communication overhead between Xplus and PostgreSQL. (By implementing Xplus inside PostgreSQL, the plan costing time can be cut down to a few seconds.) The remaining 9 minutes were spent executing subplans. 2 out of the 7 subplans took much longer to complete than the others, which is the price that Xplus pays to provide the strong guarantee in Property 2.

C.2 Comparing Xplus with Other Approaches

We now compare Xplus with the Leo and ATO controllers on the tuning task from Section 5.3: *find a plan that is 5x faster than the current plan picked by the PostgreSQL optimizer for Query 7*. As we saw in Section C.1, Xplus finds plan N_8p_{1270} for Query 7. This plan gives a speedup of 12 over PostgreSQL's plan N_1p_1, which is far higher than the requested speedup of 5.

Policy	Convergence Time (sec)	Completion Time (sec)	Number of Subplans Run
Priority	131.58	295.66	6
Round Robin	130.00	258.47	5
Rewards	130.56	294.19	6
Exploiter-Explorer	128.28	262.31	5
Explorer-Only	225.05	260.46	5

Table 8: Comparing expert-selection policies for tuning Query 7. The default Parallel Experts controller with MPL_T=2 is used

For this tuning task, the Leo controller runs into a livelock exactly like the Pure Exploiter. The best plan found by the Leo controller is N_1p_{1121} which offers a speedup of 3.2 only. Thus, the Leo controller fails the tuning task. The ATO controller generates and executes a scan query per table to gather cardinalities for all per-table filter predicates. Then, all relevant two-way joins are run to collect AC values. Based on this information, the ATO controller found N_4p_{681} as the least-cost plan. This plan offers a speedup of 4.2 only, so the ATO controller also fails the tuning task.

C.3 Internal Comparisons for Xplus

Finally, we discuss the effect of the different expert-selection policies on tuning Query 7. Table 8 summarizes the comparison among the five policies for tuning Query 7 using the Parallel Experts controller. When the Priority Policy was used, a total of 6 subplans were executed in order to collect the necessary statistics to cover all neighborhoods. The Rewards policy called the experts in a different order, but the number of executed subplans as well as the convergence and completion times were roughly the same.

The Round-Robin policy, however, was able to cover all neighborhoods by executing only 5 subplans. The reason is that Round Robin—unlike the previous two policies—consulted the Pure Explorer which helped in collecting more AC values quickly. Thus, the use of the Pure Explorer decreased the number of executed plans as well as the overall completion time. The same behavior was observed for the remaining two policies (Exploiter-Explorer and Explorer-Only) that used the Pure Explorer expert.

With the exception of the Explorer-Only policy, all other policies found the best plan at roughly the same time in the tuning process. Using the Pure Explorer alone does lead to gathering a larger number of statistics quickly and decreasing the overall number of subplans to execute. However, it does not guarantee that the statistics brought in will help in finding the best plan early on.