INUM+: A leaner, more accurate and more efficient fast what-if optimizer

Rui Wang, Quoc Trung Tran, Ivo Jimenez, and Neoklis Polyzotis

University of California - Santa Cruz
{rwang10, tqtrung, ivo, alakis}@cs.ucsc.edu

Abstract—INUM is a what-if optimization technique that efficiently estimates the cost of optimal query plans under hypothetical index configurations and can thus serve as a fast alternative to conventional what-if optimization. In this paper we introduce three crucial enhancements to INUM: a principled method to handle query plans with Nested-Loop Join (NLJ) operators (to improve estimation accuracy); a method to reduce the time to preprocess a query in the workload (to reduce setup latency); and, a method to prune the amount of information stored per query (to improve estimation efficiency). We demonstrate experimentally that these improvements make INUM 5x faster and improve median estimation accuracy by 79%. Our work extends significantly the scope of workloads and tuning problems to which INUM can be applied.

I. INTRODUCTION

Given a (query or update) statement and a hypothetical index configuration, what-if optimization produces as output an estimate for the cost of the optimal execution plan under the given index configuration. What-if optimization is widely used in index-tuning tools that need to evaluate the benefit of candidate index configurations, and so modern database systems (e.g., [1], [2], [3]) already expose this service (typically by allowing the query optimizer to be called in a special what-if mode). However, what-if optimization is an expensive operation and it is often the bottleneck in index-tuning tools, which need to evaluate a large space of hypothetical index configurations [4].

INUM [5] is a fast what-if optimization technique that can mitigate the high overhead of normal what-if optimization. INUM functions as a drop-in replacement for the DBMS what-if optimizer for the cases where there is a large number of what-if optimizations (with different hypothetical configurations) over the same statement. One such example is an index-tuning tool that performs several what-if optimizations for each statement in a training workload, each corresponding to a different index configuration within a large space of candidates. Hence, INUM is not meant to be used as a general replacement for the DBMS query optimizer, whose typical use case is to optimize each input statement for the currently materialized index configuration.

At a high level, INUM works in two phases in order to estimate the cost of the optimal execution plan for a statement \(q\) and a hypothetical configuration \(X\). The first phase is invoked only once and involves a few carefully-selected calls to the DBMS what-if optimizer in order to discover the possible set of optimal execution plans for \(q\). This set of plans is then cached and reused for any what-if optimization call involving \(q\). The second phase uses the set of cached plans to estimate the cost of the optimal plan under \(X\). Specifically, INUM transforms each cached plan to use the indexes in \(X\) as access methods, and returns the cheapest transformed plan as the optimal. Note that the second phase requires little to no communication with the DBMS optimizer and can thus execute very efficiently. As shown in the original study [5], INUM accurately estimates the cost of optimal plans and runs faster (in some cases by orders of magnitude) than conventional what-if optimization. Furthermore, INUM can work on top of any what-if optimizer and is thus portable across several systems.

While INUM represents an important step towards reducing the overhead of what-if optimization, it also has specific shortcomings that limit the scope of its applicability. Specifically, INUM has rudimentary support for plans with Nested Loop Join (NLJ) operators, which in turn leads to inaccurate cost estimates when the optimal query plan involves such operators. For example, our experimental study on TPC-DS shows that the limited support for NLJ results in poor cost estimates for 80% of the queries. Moreover, populating the cache of optimal plans may require a large number of what-if calls: polynomial in the number of relations and exponential in the number of interesting orders for a particular query. Our experimental study on TPC-DS queries shows that this number can be prohibitively high for practical applications. Lastly, INUM may cache redundant information in the set of template plans for a specific query, thus resulting in unnecessary overhead when estimating the cost of optimal plans.

Our Contributions. This paper introduces INUM+, an evolution of INUM that addresses all the aforementioned issues. Specifically, the contributions of our work are the following:

- We introduce a principled method to handle query plans with NLJ operators to improve estimation accuracy (Section III).
- We leverage the assumptions about the optimizer that are formalized in [6] to reduce set up latency when populating the cache of optimal plans (Section IV).
- We introduce pruning techniques that remove redundant optimal plans from the cache to reduce the amount of computation needed during estimation (Section V).
- We present an experimental study on TPCH and TPCDS. The results show that INUM+ runs 5 times faster than
INUUM and improves median estimation accuracy by 79%.

II. PRELIMINARIES

In this section, we review INUM [5], the fast what-if optimizer on which we develop further improvements. In the interest of space, we only consider the application of what-if optimization on queries. The extension to update statements is straightforward by modeling an update as a query shell followed by an update shell [6].

A. Basic Definitions

We consider a database comprising tables \( T_1, \ldots, T_n \). An index configuration \( X \) is a set of indexes defined over the database tables. We assume that \( X \) is a subset of a universe of candidate indexes \( S = S_1 \cup \cdots \cup S_n \), where \( S_i \) represents the set of candidate indexes on table \( T_i \). We use \( \text{cost}(q, X) \) to denote the cost of the optimal plan that evaluates query \( q \) assuming that \( X \subseteq S \) is materialized. We note that \( \text{cost}(q, X) \) is determined by the cost model of the underlying DBMS optimizer and is thus measured in some abstract cost units.

A configuration \( A \subseteq S \) is called atomic [5] if \( A \) contains at most one index from each \( S_i \). For an arbitrary index configuration \( X \), we use \( \text{Atom}(X) \) to denote the set of atomic configurations in \( X \). To simplify the presentation, we assume that each query references a specific table \( T_i \) with at most one tuple variable.

Given a query \( q \), an interesting order is a tuple ordering on the columns of an equi-join predicate, a group-by, or an order-by clause in \( q \) [7]. An index covers an interesting order if the columns of the interesting order form a prefix of the index key in the case of a B-tree index, or match exactly the index key in the case of hash index. An index configuration covers an interesting order if it contains an index that covers the interesting order.

**Running example.** In this paper, we use the following two simple queries defined on the TPC-H to demonstrate how INUM works.

\[
\begin{align*}
\text{SELECT} & \quad o.\text{totalprice} \\
\text{FROM} & \quad \text{o, customer c} \\
\text{WHERE} & \quad \text{o.custkey} = \text{c.custkey} \text{ AND} \\
& \quad \text{c.acctbal} > 0 \text{ AND} \quad \text{c.acctbal} < 100 \\
\text{SELECT} & \quad o.\text{totalprice} \\
\text{FROM} & \quad \text{o, customer c} \\
\text{WHERE} & \quad \text{o.custkey} = \text{c.custkey} \text{ AND} \\
& \quad \text{c.acctbal} > 0 \text{ AND} \quad \text{c.acctbal} < 1
\end{align*}
\]

The set of interesting orders for both queries is \( \{\text{(customer.custkey)}, \text{(orders.custkey)}\} \).

B. INUM Estimation

For each query \( q \), INUM makes a few carefully selected calls to the what-if optimizer in order to compute a set of template plans, denoted as \( T\text{Plans}(q) \). A template plan \( p \in T\text{Plans}(q) \) is a physical plan for \( q \) except that all data access operators (i.e., the leaf nodes of the plan) are substituted by “slots”. A slot represents an abstract access method over a specific table, which can be instantiated using an index or a sequential scan. Note that a slot in \( p \) may have restrictions on its sorted order. Figure 1(b) shows a template plan for query (1) that has two slots and prescribes that slot \( A \) must return tuples in sorted order of attribute \text{orders.custkey}. Here the internal subplan is the same as the internal subplan of the plan shown in Figure 1(a). (We will revisit the plan in Figure 1(a) shortly.)

Given a template plan \( p \in T\text{Plans}(q) \) and an atomic configuration \( A \), we can instantiate a concrete physical execution plan by instantiating each slot with the corresponding index in \( A \), or a sequential scan if \( A \) does not prescribe an index for the corresponding relation. Figure 1(c) shows the instantiated plan for the template plan in Figure 1(b) given the atomic configuration \( A = \{\text{orders.custkey}, \text{(customer.acctbal)}\} \).

The intuition is that \( T\text{Plans}(q) \) represents the possibilities for the optimal plan of \( q \) depending on the set of hypothetical indexes. Hence, given a hypothetical index configuration \( X \), INUM estimates \( \text{cost}(q, X) \) as

\[
\text{cost}(q, X) = \min \{\text{cost}(p, A), p \in T\text{Plans}(q), A \in \text{Atom}(X)\},
\]

where \( \text{cost}(p, A) \) is the cost of the plan instantiated from \( p \) using the atomic configuration \( A \). The remaining questions are how to compute \( T\text{Plans}(q) \) and \( \text{cost}(p, A) \), which we will address next.

Assume that \( q \) references tables \( T_1, \ldots, T_m \). To populate \( T\text{Plans}(q) \), INUM first determines the set \( O_i \) of interesting orders for table \( T_i \), for \( i \in [1, m] \). INUM also includes the “empty” interesting order in \( O_i \), to account for the indexes in \( T_i \) that do not cover an interesting order. For every member \( o \in O_1 \times O_2 \times \cdots \times O_m \), INUM makes a call to the what-if optimizer with a set of hypothetical indexes that exactly cover \( o \). The plan returned by optimizer will be converted to a template plan. For instance, for query (1), INUM invokes the what-if optimizer with a set of interesting orders \( o = \{\text{orders.custkey}\} \) and obtains the optimal plan as shown in Figure 1(a). The derived template plan is shown in Figure 1(b).

Given an atomic configuration \( A \) and a template plan \( p \in T\text{Plans}(q) \), the cost of the instantiated plan, denoted as \( \text{cost}(p, A) \), is computed as a linear expression given in

![Fig. 1: INUM plan of example query 1](image-url)
cost(p, A) = β_p + \sum_{a \in A} \gamma_{pa} \tag{4}

Here, \( \beta_p \) denotes the internal plan cost of \( p \), that is, the cost of the internal operators which does not account for the cost to access the data. The intuition is that internal costs depend solely on the volume of the data and are independent of the cost of the access methods. Each term \( \gamma_{pa} \) represents the cost to implement a slot of \( p \) using index \( a \). (The slot and the index are assumed to correspond to the same relation.) \( \gamma_{pa} \) is infinity if \( a \) does not provide the sorted order required by the corresponding slot. In all other cases, the constants in Equation 4 are derived by calling the DBMS what-if optimizer, and hence \( cost(p, A) \) is measured in the same cost model and units as \( cost(q, X) \). The implication is that INUM does not maintain a different cost model but reuses the cost model of the DBMS optimizer.

III. INDEX NESTED LOOP JOIN PLANS

In this section, we show how to extend INUM in order to handle template plans that contain Index Nested Loops Join (INLJ) operators. To motivate this extension, we first describe how INUM currently handles such operators and why the existing method is inadequate. We then present our extension that is implemented in INUM+. With some modifications, this extension can also handle Nested Loop Join operators. We did not pursue this direction, however, as plans with INLJ operators are a more common use case.

INUM Handling of INLJ. When computing \( TPlans(q) \), INUM essentially ignores any of the returned plans that contain an INLJ operator. Instead, INUM caches a special template in \( TPlans(q) \) that is derived based on the following hypothetical configuration: for each relation in the query, the configuration comprises a index that matches the interesting order on that relation and also provides all attributes referenced in the query. The intuition is that this special template represents all INLJ-based plans that could be returned based on other index configurations.

Figure 2 illustrates this process for the example query 2. Query 2 is similar to query 1, except that the selection predicate accepts fewer rows and hence an INLJ becomes an attractive option. The contents of \( TPlans(q) \) are shown in the top part of the Figure. These templates are derived by running INUM on top of the DB2 what-if optimizer, for a 10GB TCP-H data set. The template \( p_{HASH} \) corresponds to a Hash-Join plan, whereas the template \( p_{OPT} \) is meant to represent all INLJ plans and is computed as described above. Note that the slots of \( p_{OPT} \) have constraints on the ordering of tuples, e.g., slot \( C \) requires an ordering on the composite key \((c.custkey, c.acctbal)\).

Now, suppose that we invoke a what-if optimization using the hypothetical configuration \( X = \{ (orders.custkey), (customer.acctbal) \} \). The resulting instantiated plans are shown in the bottom part of the Figure. Template \( p_{OPT} \) cannot be instantiated in a plan with finite cost, since the indexes do not match the ordering constraints of the slots. Hence, the only choice is to return a cost estimate based on the instantiation of \( p_{HASH} \). However, the actual optimal plan, as returned by the DB2 what-if optimizer, mirrors the structure of \( p_{OPT} \) (i.e., uses an INLJ operator) and is 100 times cheaper. This is an example where INUM would result in a severe overestimation of the optimal query cost. Our empirical results have shown that this overestimation can occur frequently.

Our Proposal. We extend INUM to directly cache all template plans that contain INLJ operators. To do that, we have to change slightly how INUM computes \( cost(p, A) \) for a template \( p \) (which now can contain INLJ operators) and an atomic configuration \( A \). Specifically, let \( rel(a) \) denote the relation on which an index \( a \) is defined. INUM+ estimates \( cost(p, A) \) as follows:

\[
\text{cost}(p, A) = \beta_p + \sum_{a \in A, i = rel(A)} x_{pi} \gamma_{pa} \tag{5}
\]

Here, each \( x_{pi} \) is a constant that depends on the relation \( T_i \) and the template plan \( p \). The expression is similar to Equation 4, except that we scale the access cost of each slot by a slot-dependent constant. (We discuss later how these constants are computed.) The intuition follows from the logic of an INLJ operator: for each tuple of the outer, the inner is probed incurring some cost. Hence, the access cost of the inner is scaled based on the number of tuples from the outer, which is a constant for a fixed template plan.

Figure 3 illustrates this extension on example query 2. Part (a) shows the optimal plan returned by the what-if optimizer during the construction of \( TPlans(q) \) when the input configuration comprises an index on the interesting order \( o.custkey \). Whereas INUM ignores this plan, INUM+ creates a template plan as shown in part (b). This specific template \( p \) has an internal cost of 28 and the constants are set as follows: \( x_{pC} = 1 \) and \( x_{pD} = 127 \), which corresponds to the
number of tuples in the outer relation. Given the hypothetical configuration \( X = \{(orders.custkey), \{customer.acctbal\}\} \), INNUM+ instantiates the plan shown in part (c) and computes a cost estimate that is lower than the cost estimate generated by INUM. (See our previous example for Figure 2.) For this particular example, the INNUM+ estimate happens to have zero error.

The process becomes slightly more complicated when the template comprises several INLJ operators, but the overall methodology remains the same: each slot is associated with a constant \( x_{pi} \) and Equation 5 is used to compute \( cost(p, A) \), which is then factored in the estimation of \( cost(q, X) \).

We now proceed to provide a formal justification for Equation 5. We say that an operator has linear cost in a template plan \( p \) if its cost is a linear combination of the costs to access its inputs. An INLJ operator has linear cost, as its total cost depends solely on the cardinalities of the inner and outer relations. A similar case can be made for Hash-Join, Merge-Join, Group-By, Order-By, and set operators. The following lemma states that if all operators have linear cost then Equation 5 captures precisely the cost of an instantiated physical plan.

**Lemma 1:** Let \( p \) be a template plan where all operators have linear cost. Equation 5 yields the correct cost of an instantiation of \( p \) using any atomic configuration \( A \).

**Proof:** (Sketch) The proof works by induction on the tree-structure of the plan. (If the plan is originally a DAG, we perform a conceptual unfolding to a tree.) The induction shows that \( \beta_p \) comprises the internal costs of all operators that do not depend on input access costs. Each constant \( x_{pi} \) is computed based on the cost expressions of the operators that lie in the same sub-tree as the access slot.

---

**IV. SPEEDING UP INUM SPACE COMPUTATION**

In this section, we show how INUM+ computes \( TPlans(q) \) for any query \( q \), while issuing far fewer calls compared to INUM.

Our work leverages the following assumption about a well-behaved optimizer, discussed in [6]: Conceptually, the optimizer chooses the cheapest execution plan from its search space, while breaking ties in a consistent way. Given a query \( q \) and an index set \( X \), we use \( plan(q, X) \) to denote the optimal plan returned by the (well-behaved) optimizer, and \( used(q, X) \) to denote the set of indexes that is used in \( plan(q, X) \). The following result has been proven in [6].

**Lemma 2:** For any index-sets \( X, Y \) and query \( q \), if \( used(q, X) \subseteq Y \subseteq X \), then \( plan(q, Y) = plan(q, X) \).

Basically, Lemma 2 indicates that for a query \( q \), after making a what-if call using \( X \) to compute \( plan(q, X) \), we can derive the same information for other index-sets \( Y \) such that \( used(q, X) \subseteq Y \subseteq X \). INUM+ utilizes this result in order to efficiently compute \( TPlans(q) \), i.e., using information from one what-if call to deduce information about several other what-if calls.

Specifically, consider a query \( q \) that references tables \( T_1, \ldots, T_m \). Similar to INUM, INUM+ derives the set of interesting orders \( O_i \) for each table \( T_i, i \in [1, m] \). Here, we overload \( O_i \) to also represent the set of indexes, each of which exactly covers each interesting order for table \( T_i \). Recall that for each element \( o \in O_1 \times \cdots \times O_m \), INUM makes a what-if call that evaluates \( TPlans(q) \) using the index-set \( o \) in order to derive the corresponding template plan. In contrast, INUM+ uses the following algorithm to avoid making redundant calls. Let \( U \) be a set of index configurations that have been investigated so far, and \( Q \) be a sequence of index configurations that INUM+ will investigate.

1) Initialize \( U \) to be empty, and put \( O_1 \cup \cdots \cup O_m \) into \( Q \).
2) If \( Q \) is empty, stop. Otherwise, let \( X \) be the next index-configuration that is removed from \( Q \).
3) If \( X \) is present in \( U \), continue to step 2. Otherwise, make a what-if call with the index-set \( X \) and add \( X \) into \( U \).
4) For every index \( o \in used(q, X) \), put the index-set \( (X - \{o\}) \) into \( Q \). Continue to step 2.

In the worst case, \( used(Y) = Y \) for every \( Y \subseteq O_1 \cup \cdots \cup O_m \) and this results in \( 2^{O_1 \cup \cdots \cup O_m} \) what-if calls. However, in practice the optimizer only selects few indexes in the first call using \( O_1 \cup \cdots \cup O_m \), so the search space is significantly reduced.

The correctness of our approach is formalized in the following theorem:

**Theorem 1:** The INUM+ algorithm computes the same \( TPlans(q) \) as the exhaustive algorithm of INUM, under the assumption that the optimizer is well-behaved.

**Proof:** (Sketch) The INUM+ algorithm will skip a configuration \( Y \) if and only if \( used(q, X) \subseteq Y \subseteq X \) for some other evaluated configuration \( X \). According to lemma 2, \( plan(q, Y) = plan(q, X) \) and hence no plan is missed.
V. PLAN AND SLOT PRUNING

In this section, we present different techniques to improve INUM\(^+\) estimation speed. The assumption is that INUM\(^+\) will compute \(cost(q, X)\) where \(X\) is a subset of some universe \(S\) that is known a-priori. This is a reasonable assumption in practical uses of fast what-if optimization, e.g., in index-tuning tools. \(S\) represents the set of candidate indexes and \(X\) is a possible recommendation.

The high level idea of the pruning mechanisms is to remove template plans and slots, which leads to a smaller \(TPlans(q)\) where each plan may have fewer slots. Hence, estimation efficiency is improved according to equations (3) and (5).

**Best cost vs. worst cost.** Given a query \(q\) and a universe of index-set \(S\), let \(\min(p, S) = \min\{cost(p, A), A \in Atom(S)\}\), for every \(p \in TPlans(q)\). The value of \(\min(p, S)\) can be computed very efficiently using formula (4). Specifically, for each slot of the template plan \(p\), INUM\(^+\) computes the minimum value of \(\gamma_{pa}\), and then sums up these values together with the internal plan cost \(\beta_p\) as the value of \(\min(p, S)\). INUM\(^+\) prunes “redundant” template plans based on the following result.

**Lemma 3:** A template plan \(p\) can be discarded if \(\min(p, S) \geq \min\{cost(p', \emptyset), p' \in TPlans(q) - \{p\}\}\). That is, if the cost of plan \(p\) when using all possible indexes in \(S\) is no smaller than the cost of another plan without using any indexes, then this plan can never be the optimal plan to evaluate \(cost(q, X)\) for any \(X \subseteq S\).

**Removing covered plans.** The second technique removes a plan \(p\) when there exists another plan \(p'\) such that \(cost(p, A) \geq cost(p', A)\) for all atomic configurations \(A \subseteq S\).

**Lemma 4:** Consider two plans \(p\) and \(p'\) satisfying the following two conditions:

\begin{align*}
(C1) \quad & \beta_p \geq \beta_{p'}, \\
(C2) \quad & \text{For every slot of } p \text{ and every index } a \text{ that is defined on the relation of this slot, we have } \gamma_{pa} \geq \gamma_{p'a}.
\end{align*}

Then, plan \(p\) can be safely removed from \(TPlans(q)\).

**Removing useless index access costs** The optimization eliminates index access costs that should not be considered when computing \(cost(q, X)\).

**Lemma 5:** Consider an index \(a \in S\), and let \(SCAN\) denote the sequential scan operation on the relation that index \(a\) is defined. If \(\gamma_{pa} \geq \gamma_{pSCAN}\), then index \(a\) shouldn’t be considered further when instantiating template \(p\).

**Merging slots.** Our assumption so far has been that a query \(q\) references a specific table \(T_i\) at most one tuple variable for ease of presentation. If \(T_i\) has more than one tuple variables in \(q\), INUM\(^+\) exploits the following optimization to merge slots covering \(T_i\). Let \(\gamma_{pia}\) denote the cost of implementing slot \(i\) using index \(a\). We need to introduce the extra subscript \(i\), since there may be several slots corresponding to the relation on which \(a\) is defined.

**Lemma 6:** Consider a plan \(p \in TPlans(q)\) and two slots \(i\) and \(j\) of \(p\) that cover a same relation. The two slots \(i\) and \(j\) can be merged if the following condition holds for any indexes \(a, b \in S\).

\[\gamma_{pia} \leq \gamma_{pib} \text{ if and only if } \gamma_{pja} \leq \gamma_{pjb}\]

VI. EXPERIMENTAL STUDY

In this section, we present an experimental study of the INUM\(^+\) what-if optimizer. Our prototype implementation is in Java and runs over IBM DB2 (Express-C version). We test INUM\(^+\) using the 22 TPC-H queries and 103 from TPC-DS\(^+\). In both cases, we populate a 10GB database and invoke INUM\(^+\) to perform what-if optimization on the corresponding workload. All of our experiments are executed inside a one-core virtual machine, running ubuntu 10.04 with 4GB memory. The CPU of the hosting machine is Intel Core i5-3320M (2.60GHz).

**INUM\(^+\) space computation.** The first set of experiments evaluates the template-computation technique we presented in Section IV.

We measure the overhead to compute the set of template plans for each query in the test workload and compare it against the exhaustive method in the original INUM proposal. We use two metrics: the total wall-clock time to compute the template plans, and the total number of what-if calls issued to the conventional what-if optimizer (DB2 optimizer in this case). The latter provides a system-independent way to measure the efficiency of each technique.

Figure 4 shows the reduction of the number of what-if calls by INUM\(^+\) compared to that by INUM. Let \(wcount(method, q)\) denote the number of what-if calls needed for query \(q\). The x-axis represents the workload, and the y-axis represents the percentage of saving computed as follows:

\[
saving(q) = \frac{wcount(INUM, q) - wcount(INUM+, q)}{wcount(INUM, q)}
\]

The box plots show the smallest, lower quartile, median, upper quartile and largest reduction in TPC-H and TPC-DS queries. The number of what-if calls of INUM can be directly computed as \(|O_1| \times |O_2| \times \cdots \times |O_n|\). The results demonstrate that INUM\(^+\) reduces dramatically the number of what-if calls. The median reduction is 78% for TPC-H and 93% for TPC-DS. Invoking the DBMS what-if optimizer is an expensive operation (it costs 192ms per what-if call in our setting), so these savings translate directly to much higher efficiency.

Figure 5 shows the execution time of INUM\(^+\) for the same two workloads. We do not report results for INUM, as its exhaustive computation method takes too long to compute. The x-axis represents the workload and the y-axis represents the actual running time. The results demonstrate that INUM\(^+\) computes \(TPlans()\) efficiently, requiring a median of 733 ms for TPC-H queries and 1507 ms for TPC-DS queries.

**INUM\(^+\) cost estimation.** The second set of experiments evaluates the accuracy improvement of INUM\(^+\) after taking into account INLJ plans (Section III). For each workload, we ask DB2 to recommend a set of indexes \(X\) and generate 100 random subsets \(X_j, j \in [1, 100]\). For each query \(q_j\), we

\(^1\text{http://www.tpc.org/}\)
evaluate $cost(q_i, X_j)$ using DB2 query optimizer, INUM and INUM+. We use the following function to estimate the error of INUM and INUM+ with respect to DB2.

$$\text{Error}_m(q, X) = \frac{|cost_m(q, X) - cost_{DB2}(q, X)|}{\max(cost_m(q, X), cost_{DB2}(q, X))} \tag{7}$$

There are 22 TPC-H queries and 103 TPC-DS queries in the workloads. So the costs of $(22 \times 100 + 103 \times 100)$ pairs of $(q_i, X_j)$ are estimated.

Figure 6 and 7 show estimation error of TPC-H and TPC-DS respectively. The x-axis corresponds to estimation error, and y-axis corresponds to the percentage of estimations below the error given by x-axis. The median error of INUM on TPC-H is 44% and INUM+ reduced it to 6.2%. The median error of INUM on TPC-DS is 95.2% and INUM+ reduced it to 15%. The overall median error of INUM+ is 79% lower than INUM. The curves show that INUM+ is more accurate, with TPC-DS being the workload that sees the most benefit. For 32% $(q_i, X_j)$ in TPC-H and 28% $(q_i, X_j)$ in TPC-DS, INUM can not give an estimation because all plans for $q_i$ are NLJ plans and $X_j$ can’t cover the interesting order needed.

**INUM space pruning.** The last set of experiments evaluates the effectiveness of plan and slot pruning techniques we presented in Section V. For each workload $W$, we ask DB2 to recommend a set of indexes $S$. Then for every query $q_i$ we prune unnecessary plans, slots and index access costs. We generate 10000 random index subsets and measure the time needed to estimate each query using all subsets before and after pruning.

Figure 8 and 9 show the time needed to complete 10000 INUM+ cost estimation before and after pruning for each TPC-H query and selected TPC-DS query respectively. The time of pruned cost estimation also includes the time needed to apply pruning. The bottom line shows the time needed to compute $TPlans(q)$. On average we can reduce running time by 80% on TPC-H and 86% on TPC-DS.

**VII. CONCLUSION**

In this paper, we improved INUM in three crucial directions: handling of NLJ plans; reducing the number of what-if calls to preprocess each query; and, pruning the amount of information stored per query. We demonstrated experimentally that these improvements make INUM five times faster and improve median estimation accuracy by 79%.

**Acknowledgments.** This work was supported in part by NSF grant IIS-1018914 and DOE grant DE-SC0005428.

**REFERENCES**