Paper Number 159

Filtering with Approximate Predicates*

Narayanan Shivakumar, Hector Garcia-Molina, Chandra S. Chekuri
Department of Computer Science
Stanford, CA 94305.
{*shiva, hector, chekuri*}*@cs.stanford.edu*

Abstract

Approximate predicates can be used to reduce the number of comparisons made by expensive, complex predicates. For example, to check if a point is within a region (expensive predicate) we can first check if the point is within a bounding rectangle (approximate predicate). In general, approximate predicates may have false positive and false negative errors. In this paper we study the problem of selecting and structuring approximate predicates in order to reduce the cost of processing a user query, while keeping errors within user-specified bounds. We model different types of approximate predicates and their dependencies, we derive expressions for the errors of compound predicates, and we develop query optimization strategies. We also study the complexity of our optimization strategies under various scenarios, and we present an experimental case study that illustrates the potential gains achieved by optimizing queries with approximate predicates.

1 Introduction

As database systems are used in wider classes of applications, there arises a need for evaluating complex predicates. Such predicates can, for example, compare images in the database to some reference image, can identify "similar" text in a document database, can check for containment of points within regions, or can search for certain trading patterns in a stock market database. Because these predicates are often expensive to evaluate, application designers design cheaper *approximate predicates* to cut down the number of data elements that must be analyzed by the original or *ideal* predicate. For example, to check if a point is contained inside a complex region, we can first check if the point is within the region's bounding rectangle. One expects that most database points will not be in the rectangle, so the ideal containment test need only be run on a much smaller subset of points.

The bounding rectangle approximate predicate has *false-positive* errors, i.e., some data points satisfy the approximate predicate but not the ideal predicate. Other approximate predicates may have *false-negative* errors, where data values are incorrectly rejected; some predicates may have both types of errors. End users may be willing to tolerate limited errors in their results, in order to improve performance significantly.

Given a user query and a set of approximate predicates for some of the ideal predicates in the query, there are many ways in which the predicates can be combined to improve performance and to keep errors low. For instance, the output of one approximate predicate can be routed to another predicate; this could be useful if the first is a cheap predicate but has high false-positives, while the second predicate is more expensive but will remove the false-positives. Two predicates could be evaluated "in parallel" and their outputs combined, in order to reduce false-negative errors.

*Phone Number:(650) 723-3605, FAX: (650) 725-2588

In this paper we study the problem of selecting and structuring approximate predicates in order to improve the performance of a given user query. More specifically, our contributions are as follows:

- We present a model for approximate predicates, including their selectivities, costs and errors.
- We derive formulae for the selectivities, costs and errors for logical combinations of approximate predicates.
- We show how approximate predicates can be combined to answer Select-Project-Join queries containing expensive, user-defined predicates. We concentrate only on the case the `Where` clause of the query consists of a *conjunction* of built-in and expensive predicates. We defer processing arbitrary Boolean queries for future work.
- We suggest metrics for optimizing queries involving approximate predicates. We present optimization strategies for various scenarios of interest. For some scenarios, our strategies yield provably optimal plans; for others the strategies are heuristic ones. For some of the heuristic strategies, we develop approximation ratios that bound how far a solution can be from the optimal one. In all scenarios, we discuss the complexity of our strategies.
- We discuss how our strategies can be incorporated into existing query optimizers for extensible systems (that can handle complex predicates), and evaluate these strategies empirically to show the potential performance gains.

The rest of the paper is organized as follows. In Section 1.1 we present some motivating examples of complex, data-intensive applications where the use of approximate predicates may lead to significant cost savings. In Section 1.2 we consider related work. In Section 2 we formally characterize predicates, and outline the modifications required in a system catalog to maintain meta-data for these predicates. In Section 3 we characterize the space of query plans that must be considered. In Section 4 we propose a brute-force optimizer, followed by more efficient ones in Sections 5, 6 and 7. In Section 8 we evaluate our techniques and show the performance benefits in using approximate predicates.

1.1 Motivating applications

In the QBIC system, color histogram matching is an important way of computing similarity measures between images. This matching is based on a 256-dimensional color histogram and requires a 256 matrix-vector multiplication. However, QBIC employs a much faster "pre-computation" in 3D space to filter input to the more expensive histogram matching phase. Only images that pass the fast test are given to the histogram test, and only the ones that pass both tests are shown to the end user. This filtering saves substantial computational effort [9].

In our own case, we were motivated to study approximate predicates while implementing the Stanford Copy Analysis Mechanism (SCAM) [19, 25]. SCAM is a prototype of a copyright violation detection system [24, 25] that allows digital authors to find illegal copies (or fragments) of their documents in a digital library such as the web. For this, SCAM gets a feed of web documents from the Stanford BackRub webcrawler, and then stores and indexes these documents. SCAM can compute the set of documents with potential overlaps to a query document by finding the maximal common subsequences between each document and the query document. However this is expensive, since even if comparing two documents takes about a milli-second, checking one document against the 60 million estimated documents[1] in the web will take about 20 hours.

To be computationally tractable, SCAM adopts a filtering strategy: it uses some of its approximate predicates (based on word and sentence similarity [25]) to compute a set of candidate documents with "potential significant overlap" with the query document. These documents are subsequently checked by the pair-wise ideal test. This filtering strategy reduces the time to check a document to typically less than a minute (rather than 20 hours) on a SUN UltraSparc machine. SCAM's approximate predicates may have false positives and false negative errors. However, these errors appear to be tolerable when one is detecting copyright violations [24, 25].

There are many other motivating applications such as data scrubbing [16] and search problems. For instance, approximation algorithms with bounded errors have been developed for many NP-hard problems such as the minimum-cost traveling sales-person problem (TSP) [1], and for approximate searches in high-dimensional spaces [14]. Hence if a user can tolerate errors, these approximations can be used as a filter to complex ideal predicates, or to even replace the ideal predicates.

1.2 Related Work

Recently, there have been several proposals to optimize queries with expensive, user-defined predicates for extensible databases. The LDL project at MCC [7] and the Papyrus project at HP Laboratories [4] proposed viewing expensive predicates as relations and using the System R dynamic programming algorithm for join and expensive predicate enumeration. Hellerstein and Stonebraker [12] proposed *predicate migration* as a way of interleaving join and selection predicates. Chaudhuri and Shim [5, 6] recently proposed an algorithm to optimize the placement of expensive predicates in a query plan, in a System-R style query optimizer [21]. These optimizers do not consider approximate predicates and errors. The focus in our paper is on approximate predicates, and how to select the "right" subset of approximate predicates to *filter* input to the more expensive user-defined predicates, depending on the user's tolerance for errors.

The trade-off between quality of result versus time spent in computing result has been explored in the past in different contexts in relational databases. In sampling-based selectivity estimation [11, 31], sizes of query results are estimated by sampling procedures; better estimates are obtained by spending more time on sampling. Several researchers in the past have exploited filtering as a tool improve specific processes such as spatial joins [18], magic rewriting for OLAP style queries [22], image retrieval [9] and in approximating Datalog [3]. However the techniques we discuss in this paper are on a "meta-level;" we discuss how to compose a set of several approximate filters, such as the above, to optimize a user query.

[1]This is the current estimated number of pages in popular search engines such as Excite and AltaVista.

2 Characterizing predicates

We now define different types of predicates, and how to characterize them in terms of expense, selectivity, and errors. We distinguish between two kinds of predicates based on how they can be evaluated.

1. **Access predicate:** These predicates select and stream out tuples in a given relation, using some index access method. For instance, consider an index that identifies every image in the database based on its dominant color components. An access predicate to find all images with substantial yellow components can use this index, and stream out the corresponding "yellow" images.

2. **Restriction predicate:** These predicates are directly evaluated on a given tuple, rather than on a relation. For example, consider a predicate to check if a given image (tuple) has a substantial yellow component. We can implement a restriction predicate to compute the color histogram of the image, and check if the yellow component exceeds some threshold.

Let $I = \{I_1, I_2, \ldots, I_m\}$ be a set of ideal predicates. For each I_i, $1 \leq i \leq m$, we have a set of approximate predicates A_i ($= \{ A_{i,j} \}$) that can filter I_i. Let $A = \cup_{i=1}^{m} A_i$.

We now define the important characteristics of restriction predicate $A_{i,j}$ that approximates its corresponding ideal predicate I_i. Let *selectivity* $s_{i,j} = P(A_{i,j})$ be the probability that some given tuple satisfies $A_{i,j}$. It is then expected that for any input stream of t tuples, $t * s_{i,j}$ tuples satisfy $A_{i,j}$. Another important characteristic of $A_{i,j}$ is $e_{i,j}$, the expense of evaluating the predicate for each tuple, expressed in *units-per-tuple* (upts). We quantify $A_{i,j}$'s *false negative* error as $n_{i,j} = P(\neg A_{i,j}|I_i)$, which is the conditional probability that a tuple does not satisfy $A_{i,j}$, given that the tuple satisfies I_i. Similarly we quantify $A_{i,j}$'s *false positive* error as $p_{i,j} = P(A_{i,j}|\neg I_i)$, which is the conditional probability that a tuple satisfies $A_{i,j}$, given that the tuple does not satisfy I_i.

Access predicates have characteristics similar to restriction predicates. We define for access predicate $A_{i,j}$ the expense $e_{i,j}$ to be the expense of finding and streaming out tuples satisfying the predicate, normalized with respect to the number of tuples in the database. For instance, if an index on a database with 2000 images charges 1000 units to search and retrieve images with substantial yellow component, we define $e_{i,j} = 1000/2000 = 0.5$ upts. We define the selectivity $P(A_{i,j}) = s_{i,j}$ to be the fraction of output tuples to the total number of tuples in the relation. Similarly, we define $p_{i,j}$ to be $P(A_{i,j}|\neg I_i)$ and $n_{i,j}$ to be $P(\neg A_{i,j}|I_i)$.

Ideal predicates have expense and selectivity characteristics. In particular, we define s_i to be $P(I_i)$, and the expense of evaluating the predicate e_i. By definition, ideal predicates do not have false positive or negative errors.

EXAMPLE 2.1 Consider a relation with 1000 tuples. Out of all the tuples in the relation, 10 tuples satisfy ideal predicate I_1, i.e., $s_1 = 10/1000 = 0.01$. Say the expense of running I_1 on one tuple is 10,000 units.

Consider restriction predicate $A_{1,1}$ which has a per-tuple expense of 50 upts. Out of all the tuples in the relation, 107 tuples satisfy $A_{1,1}$. Out of these, 8 tuples also satisfy I_1. We can compute $s_{1,1} = 107/1000 = 0.107$, $p_{1,1} = P(A_{1,1}|\neg I_1) = (107-8)/(1000-10) = 0.1$, and $n_{1,1} = P(\neg A_{1,1}|I_1) = (10-8)/10 = 0.2$.

Next consider access predicate $A_{1,2}$ which costs 5000 units to execute using an index: 50 tuples satisfy $A_{1,2}$. Out of these, 9 tuples also satisfy I_1. We can compute $e_{i,j} = 5000/1000 = 5$, $s_{i,j} = 50/1000 = 0.05$, $p_{1,2} = P(A_{1,1}|\neg I_i) = (50-9)/(1000-10) = 0.04$, and $n_{1,2} = (10-9)/10 = 0.1$. □

2.1 Meta-data for predicates

Database systems typically maintain meta-data about tables, abstract datatypes, supported operations, selectivity and expense of predicates and other such information in system catalogs [23]. To support approximate predicates, we also need to maintain some additional meta-data:

- For each ideal predicate, we maintain the corresponding set of approximate predicates. These approximate predicates can be used to filter the ideal predicate, or can even replace the ideal predicate during query execution.
- For all approximate predicates, we maintain statistics on expense, selectivity, and expected false positive and negative errors.

How to compute and maintain statistics on expense and selectivity is well-known in the database community [6, 12, 31]. Estimating errors is less well-understood: we can use one of the following techniques to estimate errors of approximate predicates.

1. **Formal bounds:** Many approximation algorithms in theoretical computer science provide formal guarantees on error bounds. Examples of such approximation algorithms include approximate near-neighbor searching [14] and approximate minimum-cost TSP [1]. While these are the best form of estimates one could obtain, they are usually the hardest to obtain. Strict worst-case and average-case bounds are typically even harder to obtain.
2. **Probabilistic bounds:** Some techniques provide probabilistic estimates on errors, by providing confidence intervals on how closely the approximate result approximates the ideal result. Examples include sampling-based join selectivity estimation techniques [11], and on-line aggregation [13]. In general, these techniques typically require formal models of input distribution and good bounds are hard to obtain.
3. **Trained estimates:** Most error estimates are obtained the following way: the application designer constructs a small but *representative* training set or benchmark database. The query results obtained by applying the approximate predicate on the training set is compared with the results of applying the ideal predicate on the same data. The approximate predicate is then deemed to have expected errors corresponding to those obtained on the training set. Examples of systems that use this technique include information-retrieval systems [20] (where

Char. / Predicate	Type	Expense (upts)	Selectivity	False Positive	False Negative
I_1	Restriction	1000	0.1	0	0
$A_{1,1}$	Access	10	0.9	0.25	0.1
$A_{1,2}$	Access	50	0.2	0.1	0.1
$A_{1,3}$	Restriction	100	0.3	0.1	0.2
I_2	Restriction	2000	0.2	0	0
$A_{2,1}$	Restriction	500	0.5	0.2	0.1

Table 1: Example database catalog.

the errors are termed *precision* and *recall*), SCAM [24, 25] and data scrubbers [16]. These estimates are easy to obtain, and in practice turn out to be good estimates especially if the training set is carefully constructed. We discuss an example of this form of estimation in the full version of this paper [26].

4. **Sampled estimates:** In some cases it may be useful to estimate the error characteristics of approximate predicates dynamically at query time as follows. First, we choose a small *sample* of data using standard sampling techniques [17]. Then we estimate the errors of the approximate predicates on this training set, as discussed above. Now that we know the parameters of the approximate predicates, we can select the best query plan (see Sections 4 through 7), and execute the query. This form of estimation is especially useful when ideal predicates are *parameterized*, or when a set of approximate predicates are defined for a large class of ideal predicates. We present an example of such an estimation in the full version of this paper [26].

3 Space of query plans

Conventional query optimizers evaluate a variety of *query plans* for each user query (in terms of predicate placement, join orderings and index selections) before choosing the "best" plan to execute. With approximate predicates, the optimizer has to consider a much larger space of plans, since each user query can now be replaced with one of several *alternate* queries with approximate predicates. The query optimizer now has to choose the "best" plan among the set of original and alternate plans.

We now illustrate the space of plans possible in an extensible database that supports ideal and approximate predicates. Consider an example database with ten tuples. Consider the catalog information in Table 1, with meta-data about expensive predicates I_1 and I_2 along with their approximate predicates $A_{1,1}$, $A_{1,2}$, $A_{1,3}$ and $A_{2,1}$. The values listed in the table are "made-up" so as to make exposition clear, and should not be interpreted in any special way.

We use a standard *query tree* representation [28] to show the logical query plans for our examples in this section. The tree has relations at its leaves; selections, joins, projections and cross-products are placed at the tree's internal nodes [28]. In some cases, the trees may be annotated with other implementation details such as indices selected and interesting orders, but we will not use such annotations in our examples below for simplicity.

EXAMPLE 3.1 Consider the following simple `SELECT` query issued by the user: Find all tuples from table R satisfying predicate I_1.

We present in Figure 1 query trees for alternate queries along with their expenses (e), and overall false positive (p) and negative (n) errors. (In the next section, we show how to compute or estimate these values for any given query plan.)

Plan (a) is the tree for the user query that applies predicate I_1 on all tuples in table R. Plans (b) and (c) are examples of filtering input to I_1 by checking $A_{1,3}$ or $A_{1,2}$ on tuples in R before checking for I_1. Plans (d) and (e) show how approximate predicates can be *composed* using conjuncts ($ANDs$ or "$\wedge$") and disjuncts (ORs or "$\vee$") to filter input to I_1. Observe that by composing approximate predicates, we managed to (1) reduce the execution expense from 2500 in Plan (c) to 2400 in Plan (d), and (2) reduce the false negative error from 0.1 in Plan (c) to 0.01 in Plan (e). Plans (f) and (g) show how a restriction predicate such as $A_{1,3}$ can be "sequenced" (**SQN** or $\rightarrow$) on top of composed access predicates, to reduce the expense of Plans (c) and (e), at the cost of increased n errors. The **SQN** operator is similar to AND operator in terms of errors and selectivity, but differs in terms of expense. For instance, $A_{1,1} \wedge A_{1,3}$ will have a higher expense than $A_{1,1} \rightarrow A_{1,3}$ since in the former, $A_{1,3}$ is applied on all tuples while in the latter it is applied only to tuples that satisfy $A_{1,1}$. Plans (h) and (i) are similar to Plans (d) and (g) except they do not check (expensive) I_1 on the tuples: these plans have lower expected expense at the cost of potential false positives ($p > 0$).

The set of query trees we present in Figure 1 is clearly not complete, but gives the reader a flavor for the space of plans for filtering simple predicates. □

EXAMPLE 3.2 Consider a `JOIN` query that performs an equi-join between two relations R_1 and R_2 as shown in Plan (a) of Figure 2.

We present some possible query trees in Figure 2 along with expenses and false positive and negative errors incurred in executing each plan.

Plan (b) reduces the execution expense by filtering I_1 with $A_{1,2}$. Plan (c) is a more complex alternate query that filters tuples in R_1 and R_2 using some approximate predicates and finally performing I_1 after the equi-join. Note that this plan never checks tuples with I_2 and therefore $p > 0$. Clearly we can see that this set of alternate queries is not complete: in fact any of the alternate queries in Example 3.1 can be used to filter I_1, and similarly for I_2. Also we can push ideal and approximate predicates to several places in the query tree, both with respect to the equality join predicate, as well as to each other (as in expensive predicate placement [5, 12]). The example set of alternate queries we present however do illustrate the increased set of plans to be considered by a query optimizer. □

From the examples we see that a query optimizer should consider alternate queries where

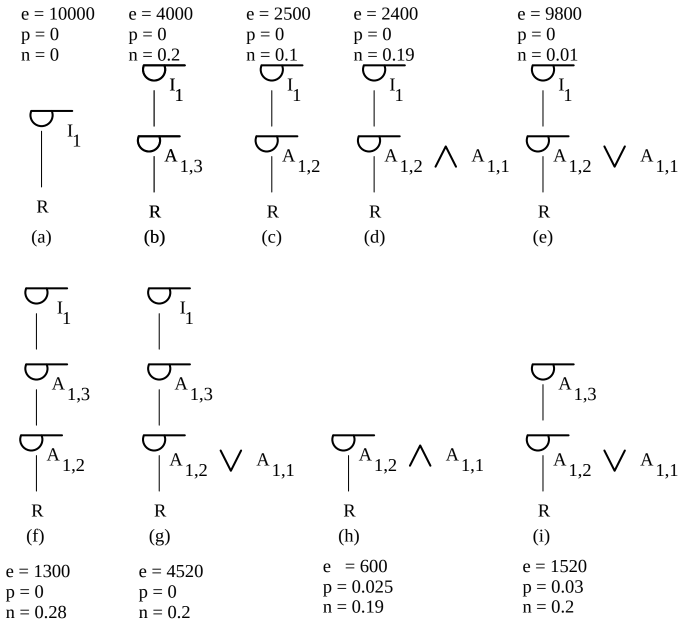

Figure 1: Some query plans for Example 3.1.

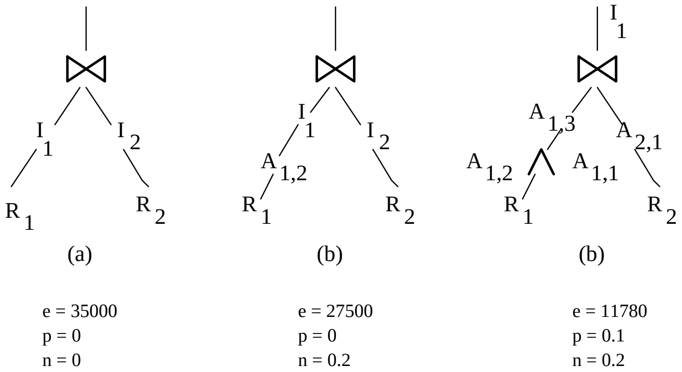

Figure 2: Some query plans for Example 3.2.

approximate predicates are composed using operators such as $\vee$, $\wedge$, $\neg$ and $\rightarrow$. In our execution model, we define *well-formed* query plans to be ones where the following compostions *cannot* be executed:

1. **Negated access predicates:** This corresponds to accessing an index to find tuples that satisfy a given predicate, and returning the rest of the tuples in the relation.

2. **Sequenced access predicates:** This corresponds to accessing an index to find tuples that satisfy a given predicate, sequencing these tuples to another index and computing a smaller set of tuples that satisfy both predicates.

Note that the second assumption does not preclude "index intersection;" our execution model does allow tuples to be retrieved from two sets of indices independently, and subsequently intersected using the $\wedge$ operator.

A query optimizer that needs to consider the space of well-formed plans becomes more complex than traditional query optimizers due to the increased number of plans. However we also see from the same examples that the potential payoffs are huge (we show this using experiments in Section 8). Hence we believe the increased complexity in building a query optimizer is a worthwhile price to pay for the potential payoffs in query execution.

3.1 Minimization measures

In this subsection we define measures to evaluate query plans for alternate queries, so an optimizer can choose the best plan to execute the given user query. In classical query optimization, the goal is to choose query plans, for a given query, with minimal query execution expense. In extensible databases with approximate predicates, two natural measures to minimize when evaluating different query plans are:

1. **Expense (*MIN-EXP*):** This measure selects the query plan with the least execution cost, irrespective of the errors, among the possible alternate queries and their physical implementations. This metric is useful when approximate predicates, that make no false negative errors, stream candidate tuples to be checked by the ideal predicate. In this case, all query plans yield correct results, so cost is the way to compare plans. This measure could also be useful in other cases, for example, if we know that all approximate predicates have acceptably low false negative errors, or if ideal predicates are *replaced* by approximate ones with acceptably low false positive rates. In these cases, errors are assumed to be low enough, and we can select plans based on cost only.

2. **Expense subject to (p, n) constraint ((p, n)-*MIN-EXP*):** In many applications, the user would like to control the quality of results returned by specifying acceptable bounds for false positive and negative errors. In such scenarios, the minimization function is expense subject to the constraint that the query plan has error estimates tolerable to the user.

Of course, *MIN-EXP* is one instance of (p, n)-*MIN-EXP* with p and n set to infinity, but we will see that we can construct more efficient query optimizers for the *MIN-EXP* measure than for (p, n)-*MIN-EXP*; hence we retain *MIN-EXP* as a minimization measure in its own right.

4 General query optimization

A query optimizer that supports approximate predicates has to choose from the space of plans illustrated in the previous section. There are a variety of approaches for this. We now present the approach we advocate in this paper, and defer a discussion of its advantages and drawbacks until after we explain the scheme. Our approach incorporates a *traditional query optimizer* (TQO), as a component. Given a logical query, the TQO performs traditional query optimization tasks such as plan enumeration, evaluating join orderings, index selections and predicate placement [28, 5, 12]. Current TQOs of course do not understand the special semantics of approximate predicates. If the TQO gets a query with approximate predicates, it treats these predicates as any other user-defined predicate. The TQO returns the best physical plan for implementing the given query, along with its estimate of the cost of executing that plan.

With our approach, we build a *wrapper* that understands approximate predicates, around the TQO. The wrapper is given a user query containing only ideal predicates, and information on the available approximate predicates. The wrapper *extends* the user query by composing some subset of approximate and ideal predicates into the query, so that the extended query has errors tolerable to the user. The wrapper then feeds this extended query to the TQO, which performs its tasks and returns the cost of executing the optimized version of the extended query. The wrapper iterates through alternate extended queries, and then chooses the alternate query whose optimized version costs the minimum to execute.

Algorithm 4.1 *NaiveWrapper for* (p,n)-MIN-EXP
Inp A: Approximate predicates; $[a_p, a_n]$: Acceptable errors
Procedure

[0] $BestPlan$:= Dummy plan with infinite cost
[1] For each $U_1 \subseteq [A_1 \cup \{I_1\}]$, $U_2 \subseteq [A_2 \cup \{I_2\}]$, $U_3 \subseteq [A_3 \cup \{I_3\}]$
[2] For each composition C_1, C_2, C_3 of predicates in U_1, U_2 and U_3 with $\vee$, $\wedge$, and $\neg$ such that the compositions are *well-formed*
[3] Construct extended query Q as $\sigma_{C_1 \wedge C_2 \wedge C_3}(R_1 \times R_2)$.
[4] If false positive and negative errors of Q are less than a_p and a_n
[5] $Q' := Q$ optimized using TQO
[6] If $cost(Q') < cost(BestPlan)$
[7] $BestPlan := Q'$
[8] Return $BestPlan$.

Figure 3: Naively wrapping any query optimizer for (p,n)-*MIN-EXP*.

We now study this wrapper approach in more detail. The wrapper we present here is clearly not efficient, and hence we call it the *NaiveWrapper*. More efficient versions will be presented in latter sections. To keep our discussion brief, we focus on the optimization of the query $[\sigma_{I_1}(R_1)] \bowtie_{I_3} [\sigma_{I_2}(R_2)]$, which is a join of two relations R_1 and R_2 with ideal predicates I_1 and I_2 to be applied on R_1 and R_2, and I_3 being the join predicate. (Generalizing to more complex queries is straightforward.)

Figure 3 shows pseudo-code for the *NaiveWrapper* for this class of join queries. The wrapper first considers all subsets of predicates in $A_i \cup \{I_i\}$, $1 \leq i \leq 3$, (Step [1]), and constructs *alternate* extended queries using $\wedge, \vee, \neg$ (Steps [2], [3]). Next, the wrapper computes the expected false and positive errors for each alternate query; the techniques for this will be covered in Sections 5 and 6. If the alternate query so produced has a tolerable error (Step [4]), it is handed to the TQO (Step [5]). (If we are using a simple MIN-EXP metric, then all such queries are handed to the optimizer.) The optimizer computes the cost of each alternate query, and the wrapper selects the alternate query with the minimum overall cost of execution.

Notice that the TQO may rearrange the predicates in order to reduce costs. For example, if C_1 is $A_{1,1} \wedge A_{1,2} \wedge I_1$, the optimizer may decide to execute them in some sequence (i.e., introducing the **SQN** operator into the tree). It is important to note that such restructuring does not change the errors of an alternate query. Thus, a query that was deemed acceptable in Step [4] will continue to be acceptable after the optimizer restructures it to reduce costs. This property makes it possible to cleanly separate the wrapper from the optimizer, and we use this property again in the wrappers that we present in the following sections.

The naive wrapper in Figure 3 exhaustively enumerates all possible alternate queries, and hence is not practical: the number of Boolean functions on n variables is 2^{2^n} when these variables are composed using $\wedge$, $\vee$ and $\neg$. So, given $|A|$ approximate predicates, the number of alternate queries produced in Steps [1] and [2] is $O(2^{2^{|A|}})$. As we mentioned earlier, in later sections we present much more efficient wrappers, at least for certain classes of queries.

The main advantage of the wrapper approach is its modularity. One can build upon existing optimizers, that codify decades of experience. Thus, to optimize queries with approximate predicates, we do not have to re-invent well known techniques for access path selections, join ordering, hash joins, and so on.

On the other hand, modularity may be a *potential* problem because current TQOs (such as those discussed in Section 1.2) assume that predicates are uncorrelated while making optimization decisions, especially during predicate placement [5, 12]. Clearly this assumption is not valid in our case since approximate predicates are correlated with the ideal predicate, and may be correlated with each other.

Of course, we can still use TQOs even if predicates are correlated, except that the resulting plans may be sub-optimal. However, in our experiments (Section 8), we observed that incorporating approximate predicates with the wrapping approach leads to significant performance improvements, despite using sub-optimal TQOs. Notice that with the wrapper approach, predicate dependencies are still handled correctly when creating alternate queries and when estimating their false positive and negative errors. Also when new TQOs are developed to correctly handle predicate correlations by future research in traditional query optimization, we can easily incorporate them immediately using our approach.

The alternative to wrappers involves the tight coupling of the enumeration of alternate plans with the optimization phase, so that alternate queries can be automatically pruned when their costs exceed the cost of another candidate alternate query. This may lead to a more efficient optimization phase, but involves modifying an existing optimizer significantly, so that error computations are incorporated into the plan evaluation process. Tight integration does not solve the predicate-interdependence issue. Thus, we would still produce sub-optimal plans, unless the optimizer is also modified to take correlations into account, which has not yet been addressed by research in query optimization.

In summary, with the wrapper approach we can immediately incorporate approximate predicates into any current query optimizer that supports user-defined predicates. For this modularity, we pay the penalty of inefficient query optimizers that do not tightly couple alternate query generation with cost-based optimization. Also, the underlying query optimizer may produce sub-optimal physical plans due to assumptions of predicate independence. However, we have observed experimentally (Section 8) that the execution time for queries drops dramatically when we incorporate approximate predicates to filter expensive predicates, despite sub-optimal physical plans.

In the next few sections, we consider how to improve the alternate query generation process, so we do not evaluate a doubly exponential number of alternate queries. In Section 5 we focus on the *MIN-EXP* measure and simple `Select` queries. In Section 6 we extend the ideas from Section 5, and develop an efficient wrapper for SPJ queries under the *MIN-EXP* measure. We subsequently show in Section 7 why optimizing for the (p, n)-*MIN-EXP* measure is hard, and then present heuristics for the (p, n)-*MIN-EXP* measure (based on our provably good wrappers for *MIN-EXP*).

5 Optimizing simple SELECT queries for *MIN-EXP* measure

In this section we consider how to build a good filter for a simple `Select` query $\sigma_{I_i}(R_1)$ for the *MIN-EXP* measure. To do this, we first need to model how approximate predicates affect each other, i.e., what is the value of an approximate predicate checking a tuple that has already been checked by another approximate predicate. In Section 5.1 we propose two models of common predicate dependencies. We then show how to construct good filters in Section 5.3.

5.1 Modeling predicate dependencies

While there are many possible ways in which predicates can depend on each other, we now consider two cases we have found common among approximate predicates in SCAM and QBIC.

1. **Local Independence (LI):** We assume that all predicates in A_i are pairwise independent but dependent on I_i. That is, $P(\wedge_{A_{i,j} \in \mathrm{A}_i} A_{i,j}) = \Pi_{A_{i,j} \in \mathrm{A}_i} P(A_{i,j})$. This is a common assumption in extensible and relational databases. This models approximate predicates that consider different attributes of incoming tuples and therefore filter tuples independent of each other.

2. **Local Conditional Independence (LCI):** We assume that all predicates in A_i are pairwise independent conditionally on I_i. That is, $P(\wedge_{A_{i,j} \in \mathrm{A}_i} A_{i,j} | I_i) = \Pi_{A_{i,j} \in \mathrm{A}_i} P(A_{i,j} | I_i)$. This assumption is strictly weaker than LI: if LI holds, LCI also holds, but the converse is not true. Under LCI, the selectivities of approximate predicates are not independent. For example, if we sequence $A_{1,1} \rightarrow A_{1,2}$, the selectivity of this filter is *not* $\sigma_{1,1} * \sigma_{1,2}$. This may be, for instance, because both $A_{1,1}$ and $A_{1,2}$ are approximating (using different techniques) the ideal predicate. So, if $A_{1,1}$ has already detected a document to be a potential copy in SCAM (using say sentence chunking [25]), that document is much more likely to be found to be a potential copy by $A_{1,2}$ (which may use word chunking [25]). However, under LCI we assume that approximate predicates make positive and negative errors independent of each other for any incoming tuple. That is, the probability that $A_{1,2}$ incorrectly identifies a document to be a copy does not depend on whether $A_{1,1}$ earlier correctly or incorrectly identified it as a potential copy. This is because the predicates are using different mechanisms that may fail in unrelated ways.

Under the above assumptions, we compute the characteristics for arbitrary predicate compositions in Section 5.2.

5.2 Propagating characteristics in a simple `Select` query

Consider the select query $\sigma_{I_i}[R_1]$. We consider alternate queries of the form $\sigma_{I_i}(\sigma_{F_i}[R_1])$, where F_i is some filter to I_1 composed only of predicates in A_i. (Including I_i in the alternate query ensures we make no positive errors, as discussed in Section 3.1.) If we trust the approximate predicates to make small false-positive errors, we could leave out the ideal predicates, but this simple variation is not discussed here.) We now show how to compute the expense, selectivity and

Operator/ Characteristics	**NOT**	**SQN**	**AND**	**OR**
$e_{i,new}$	$e_{i,j}$	$\sum_{l=1}^{k}(e_{i,j_l} * \Pi_{q=1}^{l-1}\sigma_{i,j_q})$	$\sum_{l=1}^{k} e_{i,j_l}$	$\sum_{l=1}^{k} e_{i,j_l}$
$s_{i,new}$	$1 - s_{i,j}$	$\pi_{l=1}^{k} s_{i,j_l}$	$\Pi_{l=1}^{k} s_{i,j_l}$	$1 - \Pi_{l=1}^{k}(1 - s_{i,j_l})$
$n_{i,new}$	$1 - n_{i,j}$	$1 - \Pi_{l=1}^{k}(1 - n_{i,j_l})$	$1 - \Pi_{l=1}^{k}(1 - n_{i,j_l})$	$\Pi_{l=1}^{k} n_{i,j_l}$
$p_{i,new}$	$1 - p_{i,j}$	$\Pi_{l=1}^{k} p_{i,j_l}$	$\Pi_{l=1}^{k} p_{i,j_l}$	$1 - \Pi_{l=1}^{k}(1 - p_{i,j_l})$

Table 2: Characteristics of filters constructed with LI approximate predicates.

error characteristics of the filter F_i, which we call $e_{i,new}$, $s_{i,new}$, $p_{i,new}$ and $n_{i,new}$. Following that we compute the characteristics of the complete filtered ideal query. It is important to note that the following expressions are for both access predicates, as well as for restriction predicates.

Filter for an ideal predicate: We summarize in Table 2 the characteristics of filters assuming LI predicates. We derive analogous entries for LCI predicates in Appendix A. The **AND** and **OR** operators consider a set $\{A_{i,j_1}, A_{i,j_2}, \ldots, A_{i,j_k}\} \subseteq \mathrm{A}_i$, while the **SQN** operator considers an ordered list $A_{i,j_1} \rightarrow A_{i,j_2}, \ldots, \rightarrow A_{i,j_k}$ where $A_{i,j_l} \in \mathrm{A}_i, 1 \leq j \leq k$. The **NOT** operator considers a predicate $A_{i,j}$.

For example, selectivity of filter $\vee_{l=1}^{k} A_{i,j_l}$ (second row, last column of Table 2) is

$$\begin{aligned} s_{i,new} = P(\vee_{l=1}^{k} A_{i,j_l}) &= P(\neg \wedge_{l=1}^{k} \neg A_{i,j_l}) \qquad (DeMorgan's\ Law) \\ &= 1 - P(\wedge_{l=1}^{k} \neg A_{i,j_l}) \\ &= 1 - \Pi_{l=1}^{k}(1 - P(A_{i,j_l})) \quad (LI\ assumption) \\ &= 1 - \Pi_{l=1}^{k}(1 - s_{i,j_l}) \end{aligned}$$

The other table entries for selectivities and errors are similarly derived.

The entries for filter costs are derived as follows. The cost of a **SQN** filter is simply the cost of streaming the tuples through the predicates. The cost of the **NOT** filter is simply that of its restriction predicate $A_{i,j}$ because the same work must be performed (with reversed decisions).

The cost model for unioning or intersecting streams of tuples is trickier, since it depends on the application and how the data is stored. For this paper, we assume the cost of unioning or intersecting streams to be negligible. This is the case, for example, if the access predicates generate sorted streams (sorted by say tuple-id). (The index used by $A_{i,j}$ may yield sorted tuples, or the tuples may be sorted dynamically, in which case the sort cost is included in $e_{i,j}$.) Note that information retrieval (IR) systems process unions and intersections in this fashion, and indeed the cost of the union and intersection of "inverted lists" (the sorted tuple ids) is typically negligible relative to the cost of accessing the index [20]. Other applications that use the same cost model include *mediators* that integrate a set of heterogeneous, remote databases – mediators typically assume the cost of performing unions and intersections locally at the mediator is negligible, compared to the cost of accessing and streaming back tuples from the remote sites [2, 15, 29, 30].

Of course, in some cases union and intersection costs may be significant. For example, if tuples are not sorted, we may have to use hashing to essentially execute a join. However for the rest of the paper, we continue to assume the cost of performing unions and intersections is the cumulative cost of executing the access predicates. In Appendix E we present an alternate cost model based on hashing, and we discuss its impact on the algorithms presented in the paper.

Filtered ideal query: Let e_i', s_i', n_i' and p_i' be the characteristics of the filtered ideal predicate. We then have $e_i' = e_{i,new} + \sigma_{i,new} * e_i$, $n_i' = n_{i,new}$, $p_i' = 0$. Also, we have

$$\begin{aligned} s_i' &= P(F_{i,new} \wedge I_i) \\ &= P(F_{i,new}|I_i) * P(I_i) \\ &= (1 - n_{i,new}) * s_i \end{aligned}$$

5.3 Conjunctive filters

The number of alternate queries for a simple `Select` query is doubly exponential, since restriction predicates can be composed using $\vee$, $\wedge$, $\neg$ and $\rightarrow$, and access predicates can be composed using $\vee$ and $\wedge$. In this section we restrict the operators used to compose predicates without losing optimality for the *MIN-EXP* measure, using the following observations (proofs are in Appendix B):

Observation 1 (Composing a set of access predicates)

1. *Access predicates should not be composed using* **OR**.
2. *Access predicates can be composed using the* **AND** *operator in case there is potential benefit, independent of predicate dependencies (such as LI or LCI).* □

Observation 2 (Composing a set of restriction predicates) *Restriction predicates should be composed only using the* **SQN** *operator, independent of predicate dependencies.* □

Based on Observations 1 and 2 we can safely restrict the set of query plans we need to consider to *conjunctive* filters.

Definition 5.1 (**Conjunctive Filters**) Given a set of access and restriction predicates, a conjunctive filter is produced by first composing the access predicates with the **AND** operator to form an **AND**-filter. Then the restriction predicates are composed using the **SQN** operator to form a **SQN**-filter. Finally, the *conjunctive* filter is formed by composing the **AND**-filter with the **SQN**-filter using a **SQN** operator. □

Figure 4 presents an *ConjunctiveWrapper* that only considers conjunctive filters. This wrapper is optimal for `Select` queries under the *MIN-EXP* measure. We only present the Steps that are modified from Figure 3. In Step [1] we consider all possible subsets of predicates that can be part of I_i's filter. In Step [2] we compose all the chosen predicates using **AND**. Notice that restriction

Algorithm 5.1 *ConjunctiveWrapper for simple* `Select` *query for* MIN-EXP
Inp I_i: Ideal predicate; A_i: Set of approximate predicates
Procedure
[0] $\cdots$
[1] For each $U_i \subseteq A_i$
[2] Compose $C_1 = \wedge_{a \in U_i}[a]$.
[3] Construct alternate query $\sigma_{I_i}(\sigma_{C_1}[R_1])$.
[4] – [8] $\cdots$

Figure 4: ConjunctiveWrapper for simple `Select` query for *MIN-EXP*.

predicates should be composed using **SQN**; however, they are composed with the **AND** for simplicity, since the underlying optimizer will anyway sequence the restriction predicates to minimize cost. Step [3] constructs the alternate query with the chosen predicates.

From Step [2] we see that the number of conjunctive filters is still exponential since we can choose any subset of access and restriction predicates to build a conjunctive filter. While this is significantly more efficient than the doubly exponential naive algorithm of Section 4, this approach may still be unacceptable for some applications. In the next subsection, we propose heuristics for efficient filter construction with a polynomial number of calls to the optimizer, for the *MIN-EXP* measure. These heuristics will give us *provably* good filters for LI predicates, but may give us sub-optimal plans for LCI predicates.

5.4 Conjunctive filters computable in polynomial time

We first consider how to compute a good conjunctive filter by computing good **SQN** filters for an expensive predicate. (The proofs of the following theorems are in Appendices B and C.)

Theorem 5.1 (Choosing right subset of restriction predicates)
To choose the best subset of restriction predicates to filter ideal predicate I_i for the MIN-EXP *measure, it suffices to choose all restriction predicates with rank $\frac{e_{i,j}}{1-s_{i,j}} < e_i$.* □

Theorem 5.2 (Choosing right subset of access predicates)
Suppose we construct an **AND***-filter as follows. First we rank the available access predicates by increasing value of (rank =) $\frac{e_{i,j}}{\log(\frac{1}{s_{i,j}})}$. Then, we construct the* **AND***-filter greedily: Let the* **AND***-filter contain the k-highest ranked predicates, k between 0 and the number of access predicates. If adding the $(k+1)^{th}$-highest ranked predicate into the* **AND***-filter reduces the expense of executing the alternate query, add the predicate to the* **AND***-filter. If the expense increases, the* **AND***-filter with the k-highest ranked predicates is the* **AND***-filter required. The filter so constructed is guaranteed to have expense no worse than twice that of optimal for* MIN-EXP. □

Using the results of Theorems 5.1 and 5.2, we present in Figure 5 the *LinearWrapper* for a simple `Select` query (we only present the modifications over Figure 3). LinearWrapper calls the

Algorithm 5.2 *LinearWrapper for simple* `Select` *query for* MIN-EXP
Inp A_i: Set of approximate predicates
Procedure

[0] *BestPlan* := Dummy plan with infinite cost.
[1] Compute GR_i, a set of of all "good" approximate restriction predicates for I_i, such that $(rank =)\ \frac{e_{i,j}}{1-s_{i,j}} < e_i$.
Compute GA_i, an ordered list of approximate access predicates for I_i, sorted in increasing order of $(rank =)\ \frac{e_{i,j}}{\log(1/s_{i,j})}$.
[1a] $D_1 = \wedge_{gr \in GR_i}[gr]$
[2] For each $D_2 = \wedge_{ga_k \in GA_i}[ga_k]$, $1 \leq k \leq j$, $1 \leq j \leq |GA_i|$
[3] Construct alternate query $\sigma_{I_i}(\sigma_{D_1 \wedge D_2}[R_1])$.
[4] Q' := Q optimized using underlying query optimizer
[5] If $cost(Q') < cost(BestPlan)$
[6] $BestPlan := Q'$
[7] Else Return *BestPlan*
[8] Return *BestPlan*.

Figure 5: LinearWrapper for simple `Select` query for *MIN-EXP*.

underlying optimizer m times, where m is the number of access predicates. (If there are no access predicates, the optimizer is called once.)

Recall that *NaiveWrapper* and *ConjunctiveWrapper* assumed the optimizer returns estimates of the execution cost of the optimized version of any given alternate query, and also assumed they had access to system catalogs to lookup error estimates of various approximate predicates. *LinearWrapper* in addition to these assumptions assumes the wrapper has access to system catalogs to lookup expense and selectivity characteristics of predicates.

LinearWrapper is more efficient than the doubly-exponential *NaiveWrapper* and the singly-exponential *ConjunctiveWrapper* since it is linear in the number of approximate predicates. However it is only a heuristic: it is *provably good* for LI-dependent approximate predicates under the first cost model, but may not produce optimal filters for LCI-dependent predicates. As we show in Table 5 in Appendix C, the problem of producing optimal filters for LCI-dependent predicates is NP-hard, and hence we cannot hope to find optimal, polynomial algorithms for LCI-dependent predicates.

6 Optimizing SPJ queries for *MIN-EXP*

Our *ConjunctiveWrapper* and *LinearWrapper* can easily be extended to deal with more general join queries, under the *MIN-EXP* measure. To illustrate, consider the query $Q := [\sigma_{I_1}(R_1)] \bowtie_{I_3} [\sigma_{I_2}(R_2)]$, where I_1, I_2 and I_3 are independent ideal predicates. The key idea is to treat this query, for selection of approximate predicates, as $Q := [\sigma_{I_1 \wedge I_2 \wedge I_3}(R_1 \times R_2)]$. This is equivalent to having the single predicate $I_1 \wedge I_2 \wedge I_3$ evaluated over relation $R_1 \times R_2$. (Note that this is just a conceptual way of looking at joins so we can compute the error characteristics – we are not physically implementing a cross-product!) Each approximate predicate for I_1, I_2 and I_3 can be considered approximates for $I_1 \wedge I_2 \wedge I_3$. Thus, our problem is again the selection of a good subset of approximate predicates for the single ideal predicate $I_1 \wedge I_2 \wedge I_3$.

Notice that the cost and selectivity for each approximate predicate still hold under the new ideal predicate $I_1 \wedge I_2 \wedge I_3$. For instance, consider an approximate predicate $A_{1,j}$ for I_1. Its selectivity over R_1, $s_{1,j}$, is the same as the selectivity over $R_1 \times R_2$. The false negative errors are also unchanged. For example, $n_{1,j} = P(\neg A_{1,j}|I_1)$ is identical to $P(\neg A_{1,j}|I_1 \wedge I_2 \wedge I_3)$ because $A_{1,j}$ is independent of I_2 and I_3. However, the false positive errors are changed, but can be computed from the parameters we already know. In particular, one can show that

$$p'_{1,j} = \frac{p_{1,j} * (1 - s_1) * s_2 * s_3 + s_{1,j} * (1 - s_2 * s_3)}{(1 - s_1 * s_2 * s_3)},$$

where $p'_{1,j}$ is the new error under $I_1 \wedge I_2 \wedge I_3$ and $p_{1,j}$ is the original error under I_1. We can compute false positive errors for approximate predicates of I_2, I_3 in a similar fashion. If two predicates for I_1 were LI (or LCI), they will continue to be LI (or LCI) under $I_1 \wedge I_2 \wedge I_3$. The I_1 predicates are all LI with respect to the I_2, I_3 predicates.

Once we have all the parameters for the approximate predicates, we can run *ConjunctiveWrapper* or *LinearWrapper* just as before. (Keep in mind that the wrappers only select approximate predicates to include; the underlying optimizer actually selects the join method and places the approximate and ideal predicates either before or after the join, in the order that minimizes costs, as in [5].) Similarly, if we wish to estimate the errors of a particular alternate query (to report to the user), we can use the expressions of Section 5.2 and Appendix A.

7 Minimizing (p, n)-*MIN-EXP*

In this section we consider how to minimize the (p, n)-*MIN-EXP* measure. Unfortunately, the optimal plan may no longer be a conjunctive filter in this case. We present two examples to illustrate this.

EXAMPLE 7.1 (OR Filters): Reconsider Example 3.1 from Section 3. If the maximum allowable (p, n) is $(0, 1\%)$, a good filter (and the overall optimum) would be Plan (e). This plan is however not a conjunctive filter becasue it contains **OR** operators. In general, filters using **OR** operators reduce false negative errors and hence this space could contain good plans especially when the user can tolerate few false negatives. □

EXAMPLE 7.2 (Sawtooth Filters): In Figure 6 we present an example of what we call a sawtooth filter. That filter is for an ideal test I_3, and uses restriction predicates such as $A_{3,1}$ through $A_{3,5}$. In the figure we denote tuples that are rejected by a predicate using dotted outgoing links, while tuples that satisfy a predicate flow through the unbroken outgoing link. We see that tuples flowing through the dotted outgoing link of $A_{3,1}$ are sequenced through $A_{3,2}$. The tuples that satisfy $A_{3,2}$ are then combined with those that satisfy $A_{3,1}$ to be tested by $A_{3,3}$. Similarly for tests $A_{3,3}$ and $A_{3,4}$. Again, notice that these filters are not conjunctive.

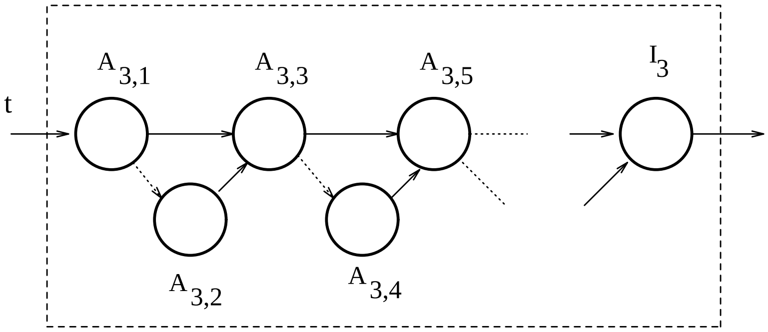

Figure 6: Example of a sawtooth filter.

A plan such as the one in Figure 6 may be useful in the following scenario. Say many tuples in a database are expected to satisfy I_3. Say $A_{3,1}$ is cheap, has high selectivity, few false positives but many false negatives, while $A_{3,2}$ is more expensive and has low false negative errors. In this case, $A_{3,1}$ acts as a fast filter to the ideal test and $A_{3,2}$ "protects" the filter by checking the $A_{3,1}$ rejects. Similarly for $A_{3,3}$ and $A_{3,4}$. □

Even though conjunctive filters may be suboptimal, it may still be useful to select the best conjunctive filter under the (p, n)-*MIN-EXP* measure. However, optimizing for (p, n)-*MIN-EXP* for conjunctive filters is NP-Hard for both LI and LCI dependent predicates (as we show in Appendix C). For some cases, we have some FPTAS (fully polynomial time approximation solutions [10]) algorithms based on dynamic programming that can compute solutions with $AR = (1+\epsilon)$. However these are only of theoretical interest and cannot be used in real systems.

Even though finding the best conjunctive filter is hard, we can still use the strategies of our previous wrappers as heuristics. For instance, the *LinearWrapper* can be modified as follows for the (p, n)-*MIN-EXP* measure: greedily insert restriction and access predicates based on their respective ranks, as long as the errors of the extended plan are tolerable. The extended plans that have tolerable errrors, will be passed to the underlying query optimizer for evaluation. The *ConjunctiveWrapper* can be similarly modified. These heuristic wrappers can be further extended so they consider a few promising **OR** filters, such as a simple **OR** of the predicates, or a sawtooth filter where high error predicates are protected. We are currently exploring and evaluating such options, but cannot report on them here due to space limitations.

In Table 3 we summarize the various wrappers we proposed in the paper, along with their complexity and some comments on their performance for *MIN-EXP* and (p, n)-*MIN-EXP* measures.

8 Experimental Results

To understand the performance of our wrappers and the quality of the plans they generate, we conducted a variety of experiments. Some of the experiments used real approximate filters (approximating the location of an address by its zip code or area code, as opposed to a precise distance

Wrapper	*MIN-EXP*	(p,n)-*MIN-EXP*	Complexity
Naive	Optimal	Optimal	Doubly exponential
Conjunctive	Optimal	-	Singly exponential
Linear	Provably good for LI	-	Linear

Table 3: Summary table of wrappers.

computation). Other experiments considered simulated queries and predicates, to evaluate performance over wider ranges of parameters. Because of space limitations, here we only summarize one of the simulator experiments. The remaining results also confirm that our scheme works very well, yielding excellent plans with relatively little effort. Readers are encouraged to read the full version of this paper [26] for more comprehensive performance results.

The goal of this one experiment was to understand how our wrappers perform when there are multiple ideal and approximate predicates. In this setup we consider a single SPJ query on randomly generated relations R_1, R_2. The query involves three ideal predicates, I_1 on R_1, I_2 on R_2, and a join predicate I_3. We assume that each ideal predicate has a number of LI approximate predicates; this number is varied in our experiments. The expense of our ideal predicates was randomly chosen between $10,000$ and $20,000$ units. We represent this uniform distribution by $U(10000, 20000)$. If an approximate predicate is an access one, its cost follows the $U(100, 1000)$ distribution. If it is a restriction predicate, its distribution is $U(10, 50)$. The selectivity of all our predicates follows $U(0.01, 1.0)$. The false positive and negative errors of approximate predicates follow $U(0.01, 0.25)$.

In this experiment we evaluated the *normalized cost* of query plans generated by our wrappers. The normalized cost of a plan is defined as the execution cost of the plan divided by the cost of the plan that uses no approximate predicates. To compute the cost of a plan, we built a simple query optimizer (TQO) based on predicate placement [5] – our optimizer considered only sort-merge and hash-partitioned joins. We expect that as more approximate predicates become available, normalized costs will drop. In our experiments, we required solutions to have zero false-positive errors (we performed experiments for other values of false-positives, but do not report them here due to space constraints). We ran the following simulations 25 times, and report the average of our results.

In Figure 7 we show the normalized cost of solutions computed by our *ConjunctiveWrapper* (*Conj*), and by the greedy extension to *LinearWrapper* proposed in Section 7 (*Linear*), as the number of approximate predicates per ideal predicate varies. In Figure 8 we plot the number of alternate query plans fed by the wrapper to the underlying query optimizer. We see in Figure 7 that as we increase the number of approximate predicates (per ideal predicate), the normalized costs drops dramatically, especially as the user is willing to accept more errors. (Notice that the vertical axis is log scale.) Also observe that while the *Linear* wrapper yields higher cost solutions compared to *Conj* wrapper, the difference is rather small. On the other hand, we see that the

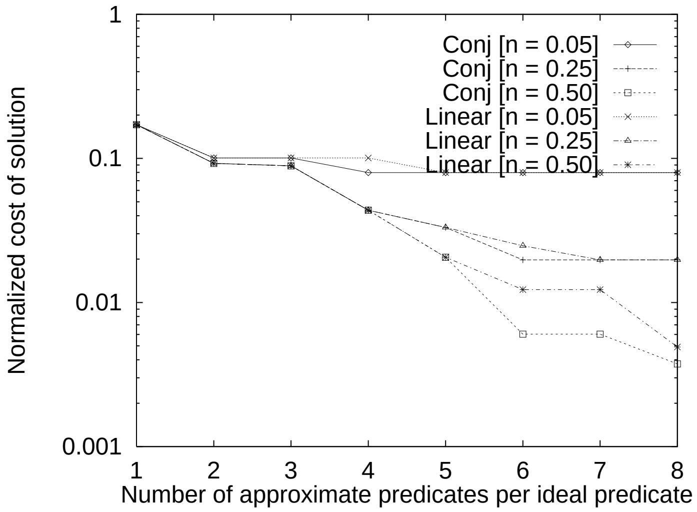

Figure 7: Quality of solutions with number of approximate predicates ($p = 0.0$).

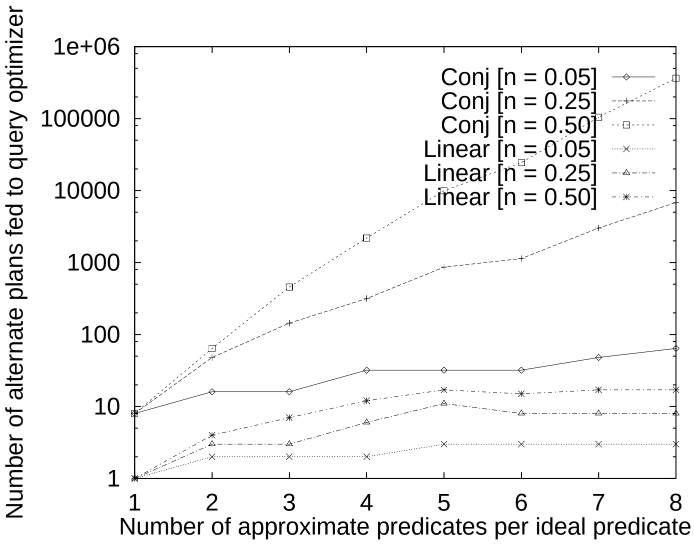

Figure 8: Number of plans with number of approximate predicates ($p = 0.0$).

number of alternate query plans handed to the optimizer under *Conj* is much larger than the equivalent number for *Linear*. Thus, we see that even though *LinearWrapper* was developed for *MIN-EXP* (which is not the metric used in this experiment), it still performed quite well under the (p,n)-*MIN-EXP* measure, with a much lower running time.

9 Conclusion

Several applications require complex and expensive predicates that may be too expensive to run on large relations. Application designers often provide simpler and computationally cheaper predicates to approximate the complex predicates. In this paper we proposed a general framework for expressing and analyzing approximate predicates, and we described how to construct alternate query plans that effectively use the approximate predicates. Our optimization strategies are provably good in some scenarios, and serve as good heuristics for other scenarios where the optimization problem is NP-hard. We also showed how to incorporate our strategies into existing query optimizers for extensible databases. Finally, we also presented experimental results that illustrate the potential performance gains, and that show *LinearWrapper* to be a very good scheme, even for the (p,n)-*MIN-EXP* metric where optimization is very hard.

In the future, we plan to consider several of the problems we identified in Section 5.4, and also some additional classes of promising filters for the (p,n)-*MIN-EXP* problem such as **OR**-filters and sawtooth filters. As we mentioned in Section 8 these classes could complement well the conjunctive filters we considered in detail in this paper. Also we plan to perform a rigorous performance study of our wrappers in the context of more general Boolean queries. We also would like to extend our techniques for more general boolean expressions. As mentioned earlier, another important challenge is to extend traditional query optimizers (TQOs), on which our scheme builds, so they do not assume pair-wise independence of expensive predicates.

Acknowledgements

We would like to thank Moses Charikar, Piotr Indyk and Suresh Venkat in the Computer Science department at Stanford for exciting and helpful discussions on this problem.

References

[1] S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of 37th Conference on Foundations of Computer Science*, Burlington, Vermont, October 1996.

[2] C. K. Chang, H. Garcia-Molina, and A. Paepcke. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4), August 1996.

[3] S. Chaudhuri and P.G. Kolaitis. Can Datalog be approximated? In *Principles of Database Systems (PODS)*, pages 86 – 96, 1994.

[4] S. Chaudhuri and K. Shim. Query optimization in the presence of Foreign Functions. In *Proceedings of 19th International Conference on Very Large Databases (VLDB'93)*, Dublin, Ireland, August 1993.

[5] S. Chaudhuri and K. Shim. Optimization of predicates with user-defined predicates. In *Proceedings of 23rd International Conference on Very Large Databases (VLDB'96)*, Mumbai, India, August 1996.

[6] S. Chaudhuri and K. Shim. Optimization of predicates with user-defined predicates. *Microsoft Research Tech. Report: MSR-TR-97-03*, March 1997.

[7] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB'89)*, pages 195 – 203, August 1989.

[8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, 1991.

[9] M. Flickner et al. Query by Image and Video Content: The QBIC System. *IEEE Computer*, pages 23 – 31, September 1995.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, New York, 1979.

[11] P.J. Haas, J.F. Naughton, S. Seshadri, and A.N. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52(3):550 – 569, June 1996.

[12] J. M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries With Expensive Predicates. In *Proceedings of ACM International Conference on Management of Data (SIGMOD'93)*, May 1993.

[13] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online aggregation. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, Tuscon, Arizona, June 1997.

[14] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards removing the curse of dimensionality. In *Stanford University, Tech Report.*, April 1997.

[15] W. Labio, R. Yerneni, and H. Garcia-Molina. Capability sensitive query processing on internet sources. In *Stanford DBGroup Technical Report*, November 1998.

[16] A. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceedings of SIGMOD 1997 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'97)*, May 1997.

[17] F. Olken. *Random sampling from databases.* Ph.D. dissertation, UC Berkeley, April 1993.

[18] J.M. Patel and D.J. DeWitt. Partition Based Spatial-Merge Join. In *Proceedings of ACM International Conference on Management of Data (SIGMOD'96)*, May 1996.

[19] P. E. Ross. Cops versus robbers in cyberspace. *Forbes Magazine*, pages 134 – 139, September 9 1996.

[20] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5), 1988.

[21] 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 P. A. Bernstein, editor, *sigmod*, pages 23–34, Boston, MA, 1979. acm.

[22] P. Seshadri and et al. Cost-based optimization for Magic: Algebra and implementation. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, May 1996.

[23] P. Seshadri, M. Livny, and R. Ramakrishnan. The case for enhanced abstract data types. In *Proceedings of the Conference on Very Large Data Bases (VLDB'97)*, pages 66–75, September 1997.

[24] N. Shivakumar and H. Garcia-Molina. SCAM: A copy detection mechanism for digital documents. In *Proceedings of 2nd International Conference in Theory and Practice of Digital Libraries (DL'95)*, Austin, Texas, June 1995.

[25] N. Shivakumar and H. Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of 1st ACM Conference on Digital Libraries (DL'96)*, Bethesda, Maryland, March 1996.

[26] N. Shivakumar, H. Garcia-Molina, and C.S. Chekuri. Filtering with Approximate Predicates. In *Stanford Database Group Technical Report*, November 1997.

Ops./ Chars.	NOT	SQN	AND	OR
$e_{i,new}$	$e_{i,j}$	$\sum_{l=1}^{k} e_{i,j_l} * [s_i * \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l}) +$ $(1 - s_i) * \Pi_{l=1}^{k}\mathrm{p}_{i,j_l}]$	$\sum_{l=1}^{k} e_{i,j_l}$	$\sum_{l=1}^{k} e_{i,j_l}$
$s_{i,new}$	$1 - s_{i,j}$	$s_i * \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l}) +$ $(1 - s_i) * \Pi_{l=1}^{k}\mathrm{p}_{i,j_l}$	$s_i * \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l}) +$ $(1 - s_i) * \Pi_{l=1}^{k}\mathrm{p}_{i,j_l}$	$s_i * (1 - \Pi_{l=1}^{k}\mathrm{n}_{i,j_l}) +$ $(1 - s_i) * (1 - \Pi_{l=1}^{k}(1 - \mathrm{p}_{i,j_l}))$
$\mathrm{n}_{i,new}$	$1 - \mathrm{n}_{i,j}$	$1 - \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l})$	$1 - \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l})$	$\Pi_{l=1}^{k}\mathrm{n}_{i,j_l}$
$\mathrm{p}_{i,new}$	$1 - \mathrm{p}_{i,j}$	$\Pi_{l=1}^{k}\mathrm{p}_{i,j_l}$	$\Pi_{l=1}^{k}\mathrm{p}_{i,j_l}$	$1 - \Pi_{l=1}^{k}(1 - \mathrm{p}_{i,j_l})$

Table 4: Characteristics of filters constructed with LCI approximate predicates.

[27] W.E. Smith. Various Optimizers for Single-Stage Production. *Naval Research Logistics Quarterly*, pages 59 – 66, 1956.

[28] J.D. Ullman. *Principles of Database and Knowledge-Base Systems (Volume 1).* Computer Science Press, 1988.

[29] V. Vassalos. In *Personal Communication*, February 1998.

[30] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *Proceedings of Very Large Databases (VLDB 97)*, August 1997.

[31] V. Poosala Y.E. Ioannidis. Histogram-based solutions to diverse database estimation problems. *Data Engineering Bulletin*, 18(3):10 – 18, 1995.

Appendix A: Propagating characteristics in a simple `Select` query (LCI)

We summarize in Table 4 the characteristics of filters assuming LCI predicates. For example, selectivity of filter $\wedge_{l=1}^{k} A_{i,j_l}$ is $P(\wedge_{l=1}^{k} A_{i,j_l})$. But we know that $P(A_{i,j}) = P(I) * P(A_{i,j}|I_i) + P(\neg I_i) * P(A_{i,j}|\neg I_i)$. Hence selectivity of filter $\wedge_{l=1}^{k} A_{i,j_l}$ (second row, third column of Table 4) is

$$\begin{aligned} s_{i,new} = P(\wedge_{l=1}^{k} A_{i,j_l}) &= P(I_i) * P(\wedge_{l=1}^{k} A_{i,j_l}|I_i) + P(\neg I_i) * P(\wedge_{l=1}^{k} A_{i,j_l}|\neg I_i) \\ &= s_i * \Pi_{l=1}^{k} P(A_{i,j_l}|I_i) + \ (1 - s_i) * \Pi_{l=1}^{k} P(A_{i,j_l}|\neg I_i) \quad (\mathit{LCI\ assumption}) \\ &= s_i * \Pi_{l=1}^{k}(1 - n_{i,j_l}) + \ (1 - s_i) * \Pi_{l=1}^{k} p_{i,j_l} \end{aligned}$$

Other entries in the table are similarly derived.

Appendix B: Proofs of observations for conjunctive filters

These are the proofs for Observations 1 and 2 from Section 5.3.

Observation 1: (Composing a set of access predicates) Access predicates should be composed only (if at all) using the **AND** operator independent of predicate dependencies.

Proof: Recall from the definition of well-formed query plans that a set of access predicates can be composed only using $\vee$ and $\wedge$. Consider two access predicates $A_{i,j}$ and $A_{i,k}$. Now compare the cost of using $A_{i,j}$ as a filter to I_i, as opposed to using $A_{i,j} \vee A_{i,k}$. Note that the latter has a higher cost since $A_{i,j}$ and $A_{i,k}$ have to be executed, and duplicate tuples may have to be removed. Also the latter is less selective compared to the former. Hence the latter composition is not useful.

Now compare the cost of using $A_{i,j}$ as a filter to I_i, as opposed to using $A_{i,j} \wedge A_{i,k}$. Note that the latter is more expensive since $A_{i,j}$ and $A_{i,k}$ have to be executed, and the tuples satisfying the two predicates have to be intersected. However the latter is also more selective than the former, i.e., the latter feeds fewer tuples to subsequent predicates. Hence the latter filter is useful in cases

Operator	LCI/ LI?	Complexity	Best		Greedy	
			Appx Ratio	Order	Appx Ratio	Order
SQN	LI	Linear	1.0	$O(1)$	1.0	$O(1)$
AND	LI	NP-Hard	$1+\epsilon$	$\tilde{O}(\frac{m^3}{\epsilon^2})$	2.0	$O(m)$
SQN	LCI		$1+\epsilon$	$\tilde{O}(\frac{q^3}{\epsilon^2})$		$O(1)$
AND	LCI	NP-Hard	$1+\epsilon$	$\tilde{O}(\frac{m^3}{\epsilon^2})$		$O(m)$

Table 5: Summary table of algorithms for *MIN-EXP*.

when the cost of performing the intersection is offset by the reduced stream of tuples for subsequent processing.

Observation 2: (Composing a set of restriction predicates) Restriction predicates should be composed only using the **SQN** operator, independent of predicate dependencies.

Proof: Consider two restrictive predicates $A_{i,j}$ and $A_{i,k}$. While the expense of $A_{i,j} \vee A_{i,k}$ is the same as that of $A_{i,j} \wedge A_{i,k}$, the latter filter is more selective than the former, thereby streaming lesser tuples to subsequent predicates.

The **SQN** and **AND** operators are equivalent in their error and selectivity characteristics, and differ only in their expense characteristic, independent of predicate dependencies. Since composing restriction predicates using **SQN** leads to lower expenses, we compose restriction predicates using **SQN**.

Appendix C: Summary of algorithms

In Table 5, we summarize how hard it is to compute optimal **AND** and **SQN**-filters, if the number of access and restriction predicates is m and q respectively. For each measure we report its complexity, i.e., is the problem NP-Hard or polynomially solvable (proofs are in the Appendix), the best *approximation* algorithm we have designed to solve these problems along with their running time order[2], as well as the approximation ratio and running time order of the greedy algorithms we present. For example, the first row of the table indicates that we can compute **SQN** filters that minimize the *MIN-EXP* measure for LI predicates with a single call to the optimizer. The greedy algorithm (Theorem 5.1) is in fact the best algorithm possible for this problem. This is stated by the approximation ratio of 1.0 shown in the table in this case. In general, an algorithm with $AR = r$ produces solutions always within a factor of r with respect to an optimal algorithm. That is, if the optimal algorithm for a problem produces a solution with cost e, an algorithm with an $AR = r$ will always produce solutions with cost $\leq e * r$ [10]. Note that some of the complexity and ratio entries in Table 5 are blank; they represent "open" problems that we plan to consider in the future.

It is interesting to notice that the problem of computing the best **AND** filter for LI predicates is NP-Hard (proved in Appendix B). Although the problem does not appear that different from the corresponding **SQN** filter one, surprisingly the complexity is very different.

Although we do not describe here in detail, we have also studied other filter optimization strategies beyond the greedy ones we now present in this paper. The performance of these other strategies is summarized under the column "Best" of Table 5. (When the greedy algorithm is optimal, the "Best" entry refers to the greedy algorithm.) We have fully polynomial time approximation schemes (FPTAS) [10] for many of our problems. For example, to compute the best **AND**-filter we have designed a dynamic programming solution with approximation ratio of $AR = 1 + \epsilon$ and running

[2] We use the standard big O and $\tilde{O}$ notation (that hides polylogarithmic factors) [8] to express running time orders.

Assumption/ Predicate	Access	Restriction
LI	$\sum_{A_{i,j}\in J_i} e_{i,j} + e_P * \Pi_{A_{i,j}\in J_i} s_{i,j}$	$\sum_{j=1}^{k+1}[e_{i,L_j} * \Pi_{l=1}^{j-1} s_{i,L_l}]$
LCI	$\sum_{A_{i,j}\in J_i} e_{i,j} + e_P * [s * \Pi_{A_{i,j}\in J_i}(1-n_{i,j})+$ $(1-s_i) * \Pi_{A_{i,j}\in J_i} p_{i,j}]$	$\sum_{j=1}^{k+1}[e_{i,L_j} * (s * \Pi_{l=1}^{j-1}(1-n_{i,L_l})+$ $(1-s_i) * \Pi_{l=1}^{j-1} p_{i,L_l})]$

Table 6: Minimization functions for *MIN-EXP*.

time $\tilde{O}(n^3/\epsilon^2)$, for any arbitrary ϵ. That is, we can get arbitrarily close to the optimal solution (AR tending towards 1) by choosing a small enough value of ϵ, at the expense of spending more time in computing the solution. Due to space limitations we do not describe these non-greedy solutions, since they are primarily of theoretical interest and not as practical and as easy to implement as the greedy solutions.

Appendix D: Goodness of *LinearWrapper*

Recall that we construct the conjunctive filter as follows.

1. *Compute* **SQN**-*filter for restriction predicates:* We now choose the "right" subset of restriction predicates, and compose then with **SQN**. Assume L_i is an ordered list ($|L| = k$) of restriction predicates for ideal predicate I_i. Let $(e_{i,L_j}, s_{i,L_j}, n_{i,L_j}, p_{i,L_j})$ denote the characteristics of the j^{th} predicate in L. The ideal predicate I_i is the last one in the sequence, so we assume it is the $(k+1)^{st}$ entry in L_i. The cost of such a filter is shown in Table 6 (third column). To illustrate how the entries are obtained, consider the case of LI predicates. If t tuples are to be filtered, the cost of executing the first predicate A_{i,L_1}, is simply $t * e_{i,L_1}$. The second predicate receives only $s_{i,L_1} * t$ tuples, and will cost $(s_{i,L_1} * t) * e_{i,L_2}$ to execute. If we continue in this fashion, sum the costs, and divide by t to obtain the per tuple cost, we get the expression in Table 6. This is the expression that should be minimized for LI predicates, if we are to choose the right subset of restriction predicates.

2. **AND**-*filter for access predicates:* We now choose the "right" subset of access predicates, and compose them with **AND**. Assume J_i ($\subseteq A_i$) is a set of access predicates on relation R, to be followed (composed using **SQN**) by some subsequent predicate P, with cost e_P.

   Table 6 (second column) gives the costs of the resulting filter, for the LI, LCI cases. To illustrate, consider the entry for LI predicates, $\sum_{A_{i,j}\in J_i} e_{i,j} + e_P * \Pi_{A_{i,j}\in J_i} s_{i,j}$ The first term ($\sum_{A_{i,j}\in J_i} e_{i,j}$) is the expense[3] incurred in executing the chosen access predicates, and ($e_P * \Pi_{A_{i,j}\in J_i}\sigma_{i,j}$) is the expense of streaming the selected tuples ($\Pi_{A_{i,j}\in J_i}\sigma_{i,j}$) through the subsequent predicate that costs e_P per tuple.

Theorem 9.1 *To choose the best subset of restriction predicates to filter ideal predicate I_i for the* MIN-EXP *measure, it suffices to choose all restriction predicates with rank $\frac{e_{i,j}}{1-s_{i,j}} < e_i$, if the restriction predicates are LI.*

Proof: The quantity $e_i/(1-\sigma_i)$ is the cost per tuple filtered out by predicate T_i. Since an ideal predicate is the final predicate in a sequence, we can think of the selectivity of the ideal predicate σ, as 0. Therefore in any optimal minimum expense filter we can use Smith's rule [27] (used for predicate placement as well) to argue that any predicate with $e_i/(1-\sigma_i) \geq e = e/(1-0)$ will be

[3] Recall from Section 2 that the expense of an access predicate is normalized with respect to the number of tuples in the database.

placed after the ideal predicate in the sequence. Since it is useless to have predicate after the ideal predicate, we conclude that no predicate with $e_i/(1-\sigma_i) \geq e$ will be used in any optimal solution.

Suppose there is a predicate T_l such that $e_l/(1-\sigma_l) < e$ and is not in the optimal solution S. The expense of the optimal solution is $\Sigma_{i=1}^{k}[e_{L_i} * \Pi_{j=1}^{i-1}\sigma_{L_j}] + \sigma' e$ where $\sigma' = \Pi_{i=1}^{k}\sigma_{L_i}$. Now consider a modified solution where predicate T_l is introduced just before the ideal predicate in the sequence S. The expense of the new sequence is $\Sigma_{i=1}^{k}[e_{L_i} * \Pi_{j=1}^{i-1}\sigma_{L_j}] + \sigma' * e_l + \sigma' * \sigma_l * e$. Since $e_l/(1-\sigma_l) < e$, it follows that $\sigma' * e_l < \sigma' * e - \sigma' * \sigma_l * e$ which implies that the expense of the modified solution is less than that of the optimal, a contradiction. The result follows. □

NP-Hardness Reductions for *MIN-EXP*

NP-Hardness of Computing AND filter for *MIN-EXP*:

Definition 9.1 (*MIN-EXP* **AND**)
INSTANCE: Set N of n pairs (e_i, σ_i) where e_i and σ_i are positive rationals, rationals e_{SF} and B.
QUESTION: Is there a subset S of $\{1,\ldots,n\}$ such that $\sum_{i\in S} e_i + e_{SF}\Pi_{i\in S}\sigma_i \leq B$? □

Definition 9.2 (**Subset Product**)
INSTANCE: Set A of n positive integers $\{a_1,\ldots,a_n\}$ and a positive integer P.
QUESTION: Is there a subset S of $\{1,\ldots,n\}$ such that $\Pi_{i\in S}a_i = P$? □

Theorem 9.2 *Computing* **AND** *filter for* MIN-EXP *is NP-Complete.*
Proof: We reduce the Subset Product problem [10] to *MIN-EXP* **AND**. Given an instance I of the Subset Product problem we create an instance of the *MIN-EXP* **AND** problem as follows. We set $e_i = \ln a_i$, $\sigma_i = 1/a_i$, $e_{SF} = P$ and $B = \ln P + 1$. We need to round the resulting numbers to some precision to make them rational. We will describe the precision required after the main idea of the reduction is shown. For any subset S of $\{1,\ldots,n\}$, let P_S denote the quantity $\Pi_{i\in S}a_i$. The reduction is essentially asking to minimize the function $\ln P_S + P/P_S$ over all subsets S. It is easy to show using calculus that the function $f(x) = \ln x + \theta/x$ achieves a minimum of $\ln\theta + 1$ at $x = \theta$. Therefore, $\ln P_S + P/P_S \leq \ln P + 1$ if and only if there exists a subset S such that $P_S = P$. Since a_i's are integers, if there is no solution to the instance I, for every S, $|P - P_S| \geq 1$ holds. Therefore it is sufficient to specify each $\ln a_i$ to a precision of $1/nP$ which takes only polynomially many bits in the size of the input. It is easy to see that the reduction can be done in polynomial time. □

Theorem 9.3 *There is a fully polynomial time approximation scheme for the* MIN-EXP **AND** *problem. There is also a simple greedy algorithm which gives an approximation ratio of* 2.
Proof: Both the algorithms are based on the similarity of the *MIN-EXP* **AND** problem to that of the Knapsack problem [10]. We will not describe the approximation scheme here but only the greedy algorithm. Assume without loss of generality that $e_{SF} = 1$ since we can divide all e_i and B by e_{SF} without changing the problem. For any subset S let E_S denote $\sum_{i\in S} e_i$ and P_S denote $\Pi_{i\in S}\sigma_i$. Suppose we knew E_{S^*} where S^* is an optimal solution. Then we can reformulate the problem as a knapsack problem where the items have volume e_i and profits $\log\frac{1}{\sigma_i}$ and the objective is to maximize the profits of the items picked with the constraint that the volume of the picked items is bounded by E_{S^*}. It is easy to see the equivalence. It is well known [10] that the greedy algorithm which picks the items in decreasing order of profit/volume while there is knapsack capacity left has an approximation ratio of 2.0. Notice that the greedy algorithm needs to know the knapsack capacity only to check when to stop and in fact the ordering of the items is independent of the knapsack capacity. In our reduction we do not know E_{S^*} but this is not a problem since we can pick the predicates greedily in decreasing order of $\log\frac{1}{\sigma_i}/e_i$ and note the value of the total cost after the inclusion of each predicate. At the end of the process we will pick the point where the least total cost is achieved and we will be guaranteed to be within a factor of 2.0 of the optimal. □

NP-Hardness of Computing (p, n)-*MIN-EXP* filters

We can easily see that the (p, n)-*MIN-EXP* **AND** filters for both LI and LCI predicates are NP-hard, since we can reduce the *MIN-EXP* **AND**-filters to the (p, n)-*MIN-EXP* **AND** problems with maximum n set to 1. We now show the reduction for the (p, n)-*MIN-EXP* **SQN** filter problem for LI predicates.

Definition 9.3 ((p, n)-*MIN-EXP* **SQN [LI]**)
INSTANCE: Set U of n pairs $(e_i, \sigma_i, \mathrm{n}_i)$ where e_i, σ_i, n_i are positive rationals. Also given rationals e, E and n.
QUESTION: Is there an ordered list L of pairs from U ($|L| = k$) such that $\Sigma_{i=1}^{k}[e_{L_i} * \Pi_{j=1}^{i-1}\sigma_{L_j}] + e_{SF} * \Pi_{j=1}^{k}\sigma_{L_j} \leq E$, and $1 - \Pi_{i=1}^{k}(1 - \mathrm{n}_{L_i}) \leq \mathrm{n}$? □

Theorem 9.4 (p, n)-MIN-EXP **SQN** *for LI predicates is NP-Complete.*

Proof: We reduce the Subset Product Problem defined earlier to (p, n)-*MIN-EXP* **SQN** filter problem as follows. Given an instance of the Subset Product problem with integers $a_1, \ldots, a_n$ and P, we set $e_i = 0$, $e_{SF} = 1$, $\sigma_i = 1/a_i$, $E = 1/P$, $\mathrm{n}_i = 1 - 1/a_i$ and $\mathrm{n} = 1 - 1/P$ for all $1 \leq i \leq n$. The reduction can clearly be done in polynomial time.

Let S be a solution to the instance of the (p, n)-*MIN-EXP* **SQN** filter problem. Since $e_i = 0$ for all i we have $\Pi_{i\in S}\sigma_i \leq E$ which implies $\Pi_{i\in S}1/a_i \leq 1/P$ or equivalently $\Pi_{i\in S}a_i \geq P$. Since S also satisfies the n constraint $1 - \Pi_{i\in S}(1 - \mathrm{n}_i) \leq \mathrm{n}$ which implies $1 - \Pi_{i\in S}1/a_i \leq 1 - 1/P$ or equivalently $\Pi_{i\in S}a_i \leq P$. From these two inequalities we conclude that $\Pi_{i\in S}a_i = P$. It is easy to see the converse that if there is a solution to the Subset Product instance then there is a solution to the (p, n)-*MIN-EXP* **SQN** instance.

This reduction also holds for LCI predicates. □

Appendix E: Alternate cost model for intersection

In database systems, access predicate intersections can be implemented as follows for a set of access predicates in A'_i. We assume that all tuples have a tuple identifier (such as RID or block ID). We could retrieve the tuples of some access predicate $A_{i,j} \in A'_i$, and build a hash table on tuple identifiers for quick lookup. In the hash table we leave additional space per tuple for counting the number of times the tuple occurs across the intersected access predicates. Then we could retrieve the tuples of other access predicates, probe the hash table, and increment the corresponding counter for each tuple. Finally we scan the hash table, and consider tuples with counters equal to the number of access predicates intersected – these tuples are the result of intersecting the access predicates.

We assume the per-tuple cost of building a hash table for tuples of $A_{i,j}$ is $b_{i,j}$, while the per-tuple cost of probing a hash table is $p_{i,j}$.

Note that the error and selectivity characteristics of filters are independent of the cost model of intersection or union. Of course the expense characteristic changes – in addition to the cost of applying the access predicate, we also add the cost of performing the intersection. The discussion in Section 5.3 remains the same, since the proofs of Observations 1 and 2 are independent of the cost model. Hence ConjunctiveWrappers are still optimal under the new cost model.

Theorem 5.2 changes since that depends on the cost model. We now present the modified version of Theorem 5.2 which computes provably good **AND**-filters.

Theorem 9.5 (Choosing right subset of access predicates under hashing model of intersection)
Suppose we construct an **AND**-*filter as follows. Choose some* $A_{i,l} \in A'_i$. *Then rank the rest of the*

available access predicates by increasing value of (rank =) $\frac{e_{i,j}+p_{i,j}}{\log(\frac{1}{s_{i,j}})}$. *Then, we construct the* **AND**-*filter greedily: Let the* **AND**-*filter contain* $A_{i,l}$ *and the* k*-highest ranked predicates,* k *between* 1 *and the number of access predicates. If adding the* $(k+1)^{th}$*-highest ranked predicate into the* **AND**-*filter reduces the expense of executing the alternate query, add the predicate to the* **AND**-*filter. If the expense increases, the* **AND**-*filter with the* k*-highest ranked predicates is one candidate* **AND**-*filter. Repeat the above procedure by choosing all* $A_{i,l} \in A'_i$. *Denote the* **AND** *filter with minimum cost from the above procedure to be* F_1. *Denote the* **AND** *filter with the access predicate that has smallest* $\frac{e_{i,j}}{\log(\frac{1}{s_{i,j}})}$ *to be* F_2. *Now choose from* F_1 *and* F_2*, the filter with the lower expense. The filter so constructed is guaranteed to have expense no worse than twice that of optimal for* MIN-EXP.

Proof: The proof is a straightforward extension of the proof for Theorem 5.2. □

Similar to *LinearWrapper*, we can now construct *QuadraticWrapper* that uses Theorems 5.1 and 9.5 – QuadraticWrapper calls the underlying optimizer $m \times (m-1)+1$ times, if m is the number of access predicates. *QuadraticWrapper* can be extended for (p, n)-*MIN-EXP*, and for SPJ queries in the same fashion as *LinearWrapper*.