Optimization of Multi-Version Expensive Predicates

Iosif Lazaridis
Donald Bren School of Information and Computer Sciences
University of California, Irvine, USA
iosif.lazaridis@gmail.com

Sharad Mehrotra
Donald Bren School of Information and Computer Sciences
University of California, Irvine, USA
sharad@ics.uci.edu

ABSTRACT

Modern query optimizers need to take into account the performance of expensive user-defined predicates. Existing research has shown how to incorporate such predicates in a traditional cost-based query optimizer. In this paper we deal with the optimization of the expensive predicates themselves, showing how their cost can be reduced by utilizing cheaper, but less accurate, versions of the predicates to pre-filter tuples. We discuss the generalized tuple handling mechanism, which processes tuples along a fixed sequence of versions, as well as adaptive approaches that either split tuple streams into groups, or make routing decisions at the individual tuple level. We identify the lower bound to the problem of evaluating a multi-version selection predicate by an ideal individualized plan (IIP), and develop an optimal generalized plan (OGP). We then show how realistic individualized or grouped schemes can produce an intermediate cost between OGP and IIP, if tuples substantially deviate from the average stream behavior. Our algorithms are tested experimentally, identifying many of the issues that arise whenever multi-version predicates are used.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Management, Performance

Keywords

query optimization, multi-version predicates, user-defined predicates, expensive methods, adaptive query processing, multimedia sensor networks, data streams

1. INTRODUCTION

Expensive user-defined predicates have been studied in the literature, e.g., [6, 5]. The goal of the existing work has been primarily to show how they can be incorporated in a traditional cost-based query optimizer. It is generally assumed that each expensive predicate has a selectivity, i.e., fraction of tuples that satisfy it, and a per-tuple cost. Using these parameters, the optimizer determines where to place the predicate in a query plan which also includes other operators. Predicates can be expensive due to data access (e.g., to load large objects from disk to memory), CPU (e.g., running a complex image processing algorithm), network latency (e.g., downloading files from the web or obtaining values from remote sensors), or other factors.

In this paper we deal with the optimization of the predicates themselves, showing that their cost can be reduced if additional, cheap, but less accurate predicates are evaluated in order to pre-filter tuples. Our work targets the per-tuple cost of expensive predicates, and is thus complementary to the existing research which aims to lower their cost by placing them intelligently in a broader query, reducing the number of times they are evaluated. Our approach is not just a tweak of a single component of a query, but may have substantial side benefits: if a predicate becomes cheaper, it might be placed lower in the query tree, thus reducing the input cardinalities of other operators, e.g., joins.

Consider the following SQL query (Q1) which uses an expensive face identifier method applied over a stream of image frames generated by a camera, joining it with a stored table containing images of criminals :

```
SELECT *
FROM   Camera, Criminals
WHERE  FaceIdentifier(Camera.image, Criminals.image)
```

Each invocation of the `FaceIdentifier` method loads the two candidate images, and extracts identifying features from the `Camera.image` frame of the `Camera` video stream, and also from the `Criminals.image` mug shot of the `Criminals` table. It then calculates the similarity of the (frame, mug shot) pair, and declares a match when some user- or method-defined threshold is exceeded.

All this effort is not always necessary; it would be a waste, e.g., to load a mug shot, if the camera frame did not contain a face, or to spend time identifying "eyes" and other facial features in the video frame if it did not contain a person. Suppose that the query writer has two additional predicates, `ObjectDetector` and `FaceDetector` which detect (respectively) foreign objects or human faces in the `Camera.image` frame. By using these methods we can reject some tuples at lower cost: if a frame does not contain an object or a face, then there is no reason to test whether or not it contains a *particular* human face of a criminal. We

SIGMOD'07, June 12–14, 2007, Beijing, China.

infer a negative result for `FaceDetector` from a negative result by either of the two predicates. On the downside, a positive answer still leaves us in doubt, and we are forced to evaluate `FaceDetector` anyway.

Our example shows that sometimes the outcome of the expensive method can be inferred by cheaper methods at lower cost; however, whenever we invoke the cheaper methods we risk getting no extra information. Thus, we should be smart about using pre-filtering methods, choosing to employ them only when they lead to an overall cost reduction.

Is it possible to get the benefit of pre-filtering without introducing new changes to the query optimizer? We could perhaps modify the `WHERE` clause of Q1 to:

```
WHERE  ObjectDetector(Camera.image, background)
   AND FaceDetector(Camera.image)
   AND FaceIdentifier(Camera.image, Criminals.image)
```

The three expensive methods would be evaluated in some order during query execution. For example, if `ObjectDetector` was placed first, we would be able to reject most empty frames, achieving the pre-filtering advantage we described. However, two problems would potentially arise:

First, the three predicates are not independent; in fact, we depend on their dependence, since this property allows us to infer the decision of the expensive method. For example, if `ObjectDetector` has a selectivity of 0.5 and `FaceDetector` has a selectivity of 0.3, then the selectivity of their conjunct may not be $0.3 \times 0.5 = 0.15$, but rather, say 0.29, if, e.g., the camera is placed at a security checkpoint and takes pictures mostly of humans passing in front of it.

The second problem is that a match can only be declared after all three versions have been evaluated. If, e.g., the security checkpoint is busy, then there is always an object, and indeed, a face in view, and we pay the cost of the cheaper versions unnecessarily. This would not be the case for a camera placed in a mostly empty corridor, where `ObjectDetector` could reject most frames, assuming, of course, that we were fortunate and the query optimizer decided to execute it in advance of the other two predicates. The standard way of arranging the predicates [7, 5] would be in order of increasing $\frac{cost}{1-selectivity}$; `ObjectDetector` has low *cost* but also high *selectivity*, so we might be unlucky and `FaceDetector` might be placed first; now, `ObjectDetector` would have no filtering power—since all faces are also objects—and would act as pure dead weight during execution.

Our paper makes several specific contributions:

- We identify the importance of optimizing multi-version predicates (MVPs) as a way to reduce the cost of expensive methods.
- We show how to choose which versions to use, depending on their per-tuple costs and selectivities; this is the Optimal Generalized Plan (OGP).
- We calculate the lower bound of the cost that can be achieved by an ideal query processor which handles tuples individually rather than sending them all along a fixed sequence of versions; this is the Ideal Individualized Plan (IIP).
- We discuss how evaluation cost intermediate between that of OGP and IIP can be achieved by schemes that either handle tuples individually or in groups. Such schemes work well when tuples exhibit substantially different-than-average behavior.
- We empirically evaluate our techniques, illustrating their good properties and identifying how performance varies in different settings of the problem.

We have motivated MVPs in terms of a video data stream processor, but in our paper we will not assume any particular type of input data or predicate type. The optimization framework is applicable whenever a predicate can be decided in different levels of accuracy/cost; this may arise in numerous settings, e.g., due to imprecise object replicas being stored at different levels of the memory hierarchy or across the network, objects at different levels of detail being stored in a single database, or lazy/eager amounts of processing being applied to the same pieces of data.

The paper is organized as follows: in Section 2 we formally define our notion of MVPs. In Section 3 we develop the Optimal Generalized Plan, discussing also an extension when version costs are not independent. In Section 4 we discuss how to reduce the cost of the OGP by individualized or grouped tuple handling. In Section 5 we report some experimental results validating the usefulness of our techniques. We cover some related work in Section 6. Finally, we conclude and present some future work in Section 7.

2. MULTI-VERSION PREDICATES

We now formally define our notion of multi-version predicates. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be selection predicates with per-tuple costs $c_1, c_2, \ldots, c_n$ with $c_1 < c_2 < \ldots < c_n$. Let $d_{t,i}$ be true iff tuple t is decided conclusively (i.e., either positively/YES or negatively/NO) by predicate λ_i. If T is the set of input tuples, then we define the M-selectivity of predicate λ_i, for $i \in \{1, 2, \ldots, n\}$ as:

Definition 1. The *M-selectivity* of predicate λ_i over set of input tuples T is $m_{i,T} = \frac{|t \in T \wedge \neg d_{t,i}|}{|T|}$.

We will note $m_{i,T}$ as m_i for conciseness, assuming some input set T. For convenience, we can also define an input pseudo-version λ_0 with $c_0 = 0$ and $m_0 = 1$ and $\forall t : \neg d_{t,0}$. We are interested in the case that $m_1 > m_2 > \ldots > m_n$, i.e., when higher-cost versions are able to decide more tuples. In Q1, the three versions are λ_1=`ObjectDetector`, λ_2=`FaceDetector`, and λ_3=`FaceIdentifier`, and their M-selectivities might be, e.g., 0.5, 0.3, and 0.05, if 50% of frames contain a foreign object, 30% of them contain a face and 5% of them contain a criminal's face. To save space, we will note sequences like $a_k, a_{k+1}, \ldots, a_n$ as $\{a_i\}_k^n$. We make the assumption of *honesty* about our versions:

Definition 2. A multi-version predicate $\{\lambda_i\}_1^n$ is *honest* if whenever a version λ_i decides a tuple as YES or NO, then all subsequent versions (λ_j with $j > i$) also decide it as YES or NO respectively.

The assumption of version honesty is necessary to achieve the filtering power of the cheaper versions. Without it, whenever `FaceDetector` claimed that there was no face in a frame, and by implication no face of a criminal, it would be possible that `FaceIdentifier` would find a positive match; hence, we would always have to use `FaceIdentifier` unless we were willing to tolerate some false positives/negatives

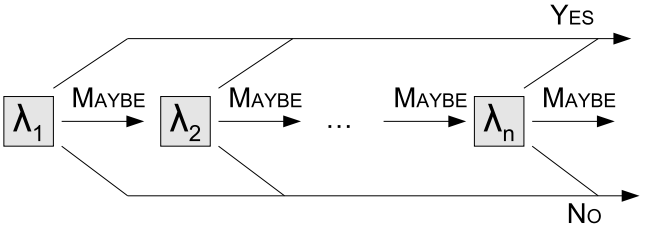

Figure 1: A train of selection versions

whenever the cheaper version was being "dishonest." We can also define *semi-honest* versions, which behave honestly for most but not necessarily all tuples. In our paper we will focus on honest versions, but we illustrate the issues associated with semi-honest ones in Section 5.

A natural way of using multiple versions is to organize them in a "train" as shown in Figure 1. Early versions cull tuples, either sending them to the output or rejecting them, and undecided (MAYBE) tuples continue along the train until they reach the final version λ_n. Note that we allow λ_n to also produce some undecided tuples; in our video stream example these may be borderline face matches. MAYBE tuples may be either added to the answer or not depending on desired semantics ("subset of the answer": not added to the answer [12]; "variable recall/precision": added [9]).

Filtering Modalities: In the Q1 example, cheap versions can never place a tuple in the YES path, since they are unable to confirm that a tuple satisfies the `FaceIdentifier` predicate: their filtering power is only due to their ability to reject some tuples. Consider a different query (Q2) asking for "Which cameras contained 'Bob' between 08:00 and 08:30." Rather than retrieving a huge amount of video from our video archive we can extract a few frames from each camera stream, say 1% of the total data (λ_1), or extract longer segments around the key frames, amounting to 10% of the data (λ_2). Using either λ_1 or λ_2 we can confirm Bob's presence but we cannot reject it, since he might be present in the remaining (99% or 90%) of the video data.

It is also possible that cheaper versions may both confirm and reject tuples, sending them to either the YES or NO paths. Consider a hierarchical caching scheme where highly dynamic values generated in a sensor network (λ_3) are approximated by intervals at the wireless access point (λ_2) and by wider intervals at the central database (λ_1); this might be useful to reduce data transmission by avoiding updates when the value remains within its interval approximation. If the query (Q3) asks for values ≥ 42, then λ_1 interval $[10, 20]$ could be rejected, interval $[50, 60]$ could be accepted, while interval $[40, 45]$ would remain MAYBE.

Definition 3. The cost of a multi-version predicate $\{\lambda_i\}_1^n$ if all versions are organized in a train (Figure 1) is:

$$C_{\text{ALL}}(\{\lambda_i\}_1^n) = c_1 + \sum_{i=2}^{n} m_{i-1} c_i \tag{1}$$

The above assumes that all tuples are routed along exactly the same sequence of versions, exiting the train when they are decided. Note that by "cost" we refer to the cost per each input tuple processed by the multi-version predicate. There is room for improvement beyond this "generalized" method. We can derive a lower bound for the problem by noting that ideally each tuple will be handled only by a single version, namely the one that is able to decide it as YES or NO with minimum cost. We define:

Definition 4. The *sufficient version* for a tuple t is:

$$s_t = \begin{cases} \lambda_i & \text{if } \exists i \in \{1, 2, \ldots, n\} : d_{t,i} \wedge \neg d_{t,i-1} \\ \dashv & \text{otherwise} \end{cases} \tag{2}$$

Note that by $\dashv$ we note the event where a tuple has no sufficient version, i.e., it remains undecided even at the last (λ_n) version. If $s_t = \dashv$ then ideally *no* versions should be evaluated over t, because none of them will be able to resolve it. Using the concept of a sufficient version we can calculate the cost of an Ideal Individualized Plan (IIP):

THEOREM 1 (COST OF IIP). *The cost of the ideal individualized plan is:*

$$C_{IIP}(\{\lambda_i\}_1^n) = (1 - m_1)c_1 + \sum_{i=2}^{n} (m_{i-1} - m_i)c_i \tag{3}$$

PROOF. Ideally, only the sufficient version s_t of each tuple should be evaluated for it. If a more expensive version is evaluated, then it will be able to decide it (because of honesty) but at greater cost. If a cheaper version is evaluated then by definition of the sufficient version it will not be able to decide the tuple. Hence, the per-tuple cost of the IIP plan will be $C_{IIP}(\{\lambda_i\}_1^n) = \frac{\sum_{i=1}^{n} c_i |\{t \in T \wedge s_t = \lambda_i\}|}{|T|} = \sum_{i=1}^{n} c_i \frac{|\{t \in T \wedge s_t = \lambda_i\}|}{|T|} = \sum_{i=1}^{n} c_i (\frac{|\{t \in T \wedge d_{t,i}\}|}{|T|} - \frac{|\{t \in T \wedge d_{t,i-1}\}|}{|T|}) = \sum_{i=1}^{n} c_i (\frac{|\{t \in T \wedge \neg d_{t,i-1}\}|}{|T|} - \frac{|\{t \in T \wedge \neg d_{t,i}\}|}{|T|}) = \sum_{i=1}^{n} c_i (m_{i-1} - m_i)$ which proves the theorem, since $m_0 = 1$. □

Intuitively, λ_1 is able to decide $1 - m_1$ fraction of tuples, λ_2 can decide $m_1 - m_2$ and so on. Clearly this ideal cost is unrealizable, since we cannot a priori know for a particular tuple the identity of the version which decides it with minimum cost; in Section 4 we will show how more realistic individualized tuple handling mechanisms can be achieved.

A geometrical interpretation of the cost is seen in Figure 2 where the per-tuple cost and M-selectivity is plotted on a (c, m) plane. The shaded area represents the MVP evaluation cost: FINAL indicates a plan where only the final version is used (λ_n), i.e., additional cheaper versions are not exploited. ALL represents a plan where all versions are used $(\lambda_1, \lambda_2, \lambda_3)$; the number of times that the cost represented by the various areas must be paid is indicated. The IIP plan avoids paying costs multiple times, according to Equation 3.

FINAL is the standard way of evaluating the expensive predicate; no advantage is taken of the less expensive methods. ALL is an improvement over the standard way: it uses the additional methods and places them in the correct order of decreasing M-selectivity; however, it has the potential pitfall of paying the cost of all versions if the cheaper ones produce many MAYBE tuples. In the next section we will show how to produce a better plan by deciding which versions to use; in Section 4 we will discuss more adaptive schemes that try to reduce the cost even further.

3. OPTIMAL GENERALIZED PLAN

Given n different predicate versions there are 2^n different subsets of them which could be evaluated over the input

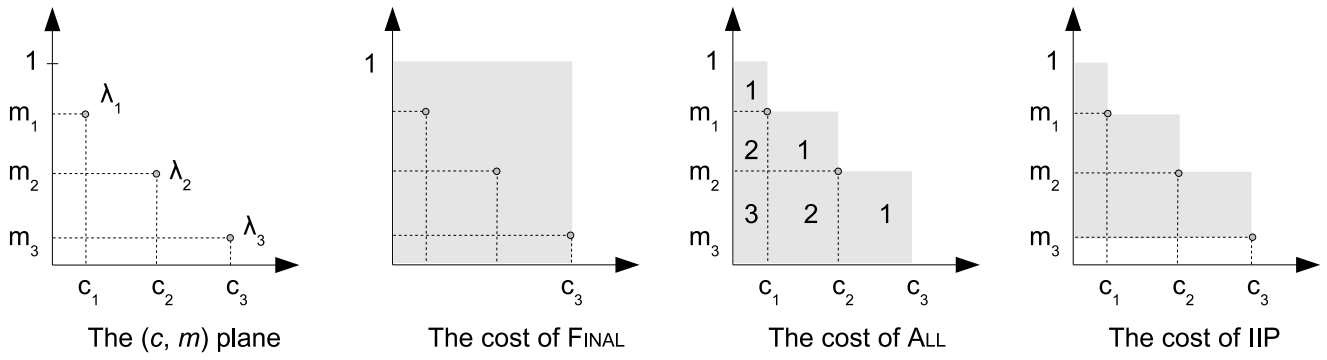

Figure 2: Geometrical interpretation of cost

relation or stream. Since we require that the exact answer should be produced, we must always evaluate λ_n. Let e_i be true iff λ_i is evaluated. We can compute the optimal subset out of the 2^{n-1} possible choices using the following theorem.

THEOREM 2 (OPTIMAL GENERALIZED PLAN). *Let $S = \{\lambda_i\}_1^n$ be a sequence of selection predicate versions. Let $S_l = \{\lambda_i\}_l^n$. Let $C_{k,l}$ be the optimal cost incurred by versions in S_l given that $k < l \wedge e_l \wedge \nexists j : (k < j < l \wedge e_j)$. Then:*

$$C_{k,l} = \begin{cases} m_k c_l & \text{if } l = n \\ \min\{m_k c_l + C_{l,l+1}, C_{k,l+1}\} & \text{otherwise} \end{cases} \quad (4)$$

PROOF. λ_k is the "previous" version of λ_l; it is the case that λ_k has been evaluated, whereas all versions between λ_k and λ_l have not. $C_{k,l}$ represents the optimal generalized cost incurred at versions starting at λ_l, given that the previous version (i.e., before λ_l) is λ_k. Clearly, some of the versions in S_l will be included and some of them will not be. More specifically, either λ_l will be included or not. If it is included, then the cost incurred in S_l will be $m_k c_l + C_{l,l+1}$ because version λ_l will be evaluated over fraction m_k of tuples coming from the previous version. Otherwise, the cost incurred in S_l will be $C_{k,l+1}$ because no cost is incurred at λ_l and hence, the remaining cost will be incurred in S_{l+1} given that k is the previous version. We rely on the observation that the cost paid at versions starting at λ_l depends only on the previous version λ_k and not any versions cheaper than λ_k. The border condition is for $l = n$, i.e., when λ_l is the final version, which must always be evaluated. □

The algorithm to compute the optimal generalized plan is given in Figure 3. It uses a *skip* table which keeps track of the decision whether to evaluate or skip version λ_l given that the previous version is λ_k. The algorithm outputs the *cost* of the optimum generalized plan as well as *included*, a set of the integer indices of the versions included in the plan. The time and space complexity of OGP is $O(n^2)$ which is reasonable, since we anticipate that n will not be very large in most practical problems. Its good performance in practice is reported in Section 5.

The cost of the OGP for the entire sequence is $C_{0,1}$ as stated in the following, because it is the optimal cost incurred in versions in $S_1 = \{\lambda_i\}_1^n$, i.e., the entire S, given that λ_0 is the previous version.

```
(1)  function OGP
(2)  INPUT: real m[1..n], c[1..n];
(3)  OUTPUT: real cost; set of integer included;
(4)
(5)  if (n=1) {
(6)    cost ← c[n];
(7)    included ← {n};
(8)  }
(9)  else {
(10)   real C[0..n−1][1..n];
(11)   boolean skip[0..n−1][1..n];
(12)   for (k ← 0 to n−1) {
(13)     C[k][n] ← m[k]*c[n];
(14)     skip[k][n] ← false;
(15)   }
(16)   for (l ← n−1 down to 1)
(17)     for (k ← 0 to l−1) {
(18)       real notSkipCost ← m[k]*c[l]+C[l][l+1];
(19)       real skipCost ← C[k,l+1];
(20)       C[k][l] ← min{notSkipCost, skipCost};
(21)       skip[k][l] ← (notSkipCost>skipCost);
(22)     }
(23)   cost ← C[0][1];
(24)   included ← {n};
(25)   k ← 0;
(26)   for (l ← 1 to n−1)
(27)     if not skip[k][l] {
(28)       included ← {l} ∪ included;
(29)       k ← l;
(30)     }
```

Figure 3: Computing the Optimal Generalized Plan

COROLLARY 1 (COST OF OGP). *Let $S = \{\lambda_i\}_1^n$ be a multi-version predicate, then the cost of the Optimal Generalized Plan over S is:*

$$C_{OGP}(\{\lambda_i\}_1^n) = C_{0,1} = C_{\text{ALL}}(\{\lambda_i | \lambda_{\in} S \wedge i \in \textit{included}\}) \quad (5)$$

The workings of OGP are illustrated in the following.

Example 1. Consider the 3-version predicate of Table 1. The cost of the OGP is calculated in Table 2. For example, entry $C_{2,3}$ is the cost incurred at operator λ_3 given that the previous operator is λ_2. Hence, $m_2 = 0.3$ fraction of tuples is input into λ_3, and a cost of 100 is paid per-tuple. Hence,

	λ_1	λ_2	λ_3
c_i	1	50	100
m_i	0.5	0.3	0.01

Table 1: Costs and M-selectivities of a 3-version predicate

	1	2	3
0	$51 = \min\{1+50, 80\}$	$80 = \min\{50+30, 100\}$	100
1		$50 = \min\{25+30, 50\}$	50
2			30

Table 2: Conditional cost C table for example of Table 1. Row k, column l represents $C_{k,l}$

$C_{2,3} = 0.3 \cdot 100 = 30$. Entry $C_{1,2}$ is the cost incurred at versions λ_2, λ_3 given that the previous operator is λ_1. It is either the case that λ_2 is used or not. If it is used, then it costs $m_1 c_2 = 0.5 \cdot 50 = 25$, and then λ_3 has λ_2 as its previous operator, hence cost $C_{2,3} = 30$ must be added (total 25+30). Alternatively, λ_2 is not used, and then the cost is equal to $C_{1,3} = 50$. Since, by definition $C_{2,3}$ must be the generalized optimum cost, we take the $\min\{25+30, 50\} = 50$. Filling out all entries in the table, we conclude that $C_{0,1} = 51$, and the optimum sequence is λ_1, λ_3. We can compare this with the generalized cost of the sequence $\lambda_1, \lambda_2, \lambda_3$ which is $C_{\text{ALL}}(\lambda_1, \lambda_2, \lambda_3) = 1 + 0.5 \cdot 50 + 0.3 \cdot 100 = 56$, which is greater. We can also calculate the ideal individualized cost $C_{IIP}(\lambda_1, \lambda_2, \lambda_3) = (1-m_1)c_1 + (m_1 - m_2)c_2 + (m_2 - m_3)c_3 = (1-0.5) \cdot 1 + (0.5-0.3) \cdot 50 + (0.3-0.01) \cdot 100 = 39.5$, which, as expected is lower.

3.1 Shared Cost

So far, we have assumed that handling a tuple has a fixed cost for each version. Sometimes this assumption is not correct, or is too pessimistic. In particular, if a version λ_k cannot resolve a tuple t, not all c_k cost paid for t is wasted, but some of it may be "used" to reduce the cost of versions with $\lambda_{k'} : k' > k$.

Consider, the `ObjectDetector` and `FaceDetector` examples from the Introduction. The `FaceDetector` method may replicate the effort of the `ObjectDetector` method to identify the location and extent of the face in the image. If this information was not lost but could be shared between the two methods, then the cost of `FaceDetector` would be reduced. Thus, it is possible to reduce the cost of an expensive predicate not only by pre-filtering tuples, but also by utilizing information computed by previous versions.

Similarly, in the hierarchical scheme of Section 2, the cost of obtaining a value from a sensor may depend on whether the value was first sought from the access point; if it was, then the communication cost with the access point may not be paid anew if the access point can be tasked to request the value from the sensor; only the cost paid to look up the value in the access point's database would have been wasted.

In general, whenever a version is evaluated, all previously evaluated versions may help alleviate part of the cost. Thus, our *OGP* procedure should be modified, since to compute the optimal cost starting at version λ_l can no longer be achieved knowing only the identity of the previous version, λ_k, but is different for each of the 2^{l-1} different combinations of previous versions.

We can exploit shared costs where available if it is the case that c_l is function of only the previous version λ_k. In our hierarchical caching example, if the access point has been contacted, then the cost of retrieving a value from the sensor is reduced by the amount necessary to communicate with the access point; whether or not the cheapest version (lookup in the server database) has been evaluated first does not reduce the cost any further. Similarly, in Q2 from Section 2, if a 10% segment has been processed, then λ_3 can avoid processing it, regardless of whether the system initially tried to identify "Bob" in the few key frames extracted by λ_1.

Whenever we can compute a version's per-tuple cost with knowledge of onlythe previous version, we can employ the same *OGP* procedure, with slight modification. Line (2) should now accept c[1..n][0..n-1] as input, such that c[l][k] denotes the per-tuple cost paid at λ_l given that λ_k has been previously evaluated. Line (6) should be modified to replace c[n] with c[n][0] and line (18) to replace c[l] with c[l][k].

In Section 5 we will see how shared costs affect optimization decisions, confirming the basic intuition that as a greater fraction of the cost paid at a version can be subtracted from the cost paid at more expensive ones, it becomes increasingly attractive to use the cheaper versions.

4. ADAPTIVE TUPLE HANDLING

We will now discuss adaptive tuple handling schemes that do not process the same sequence of versions for all tuples of the input. First, we will discuss an *Individualized* scheme which can be employed when the probability of a tuple being decided in the different versions can be estimated; then, we will discuss a *Tuple Grouping* scheme that does not model these probabilities explicitly, but identifies attributes that correlate with cost and groups tuples based on those attributes, optimizing the resulting groups separately.

4.1 Individualized Handling

Lower cost than OGP can be achieved up to the bound dictated by Equation 3 for IIP. Intuitively, since we cannot a priori know where each tuple will be decided, we can estimate the probability that it will be decided (resolved) in any of the different versions.

Definition 5. Let $e_{t,i}$ be true if version λ_i has been evaluated over tuple t. Then we define for tuple t the *resolution probability* by version λ_j given that λ_i has been evaluated as $P_{j,i}(t) = Pr(d_{t,j} | e_{t,i})$.

As an example, consider Figure 4 where a query asks for values ≥ 0 and three λ_1-intervals a, b, c of length 2 are MAYBE with respect to this query. If λ_2-intervals are of length 0.5, then we can compute the probability of the tuples being decided by the λ_2 intervals as $P_{2,1}(a) = 0.83, P_{2,1}(b) = 0.67, P_{2,1}(c) = 0.83$. If λ_3-intervals are of length 0.25, we can similarly compute $P_{3,1}(a) = P_{3,1}(b) = P_{3,1}(c) = 0.86$.[1]

This example illustrates two things: (i) as expected, tuples have a higher probability of being decided if they are sent to more expensive and more accurate versions, e.g., $P_{2,1}(b) = 0.67 < P_{3,1}(b) = 0.86$), and (ii) different tuples have potentially a different probability of being decided (resolved) by a single version, e.g., $P_{2,1}(a) = 0.83 > P_{2,1}(b) =$

[1]We make the assumption that the λ_2, λ_3 intervals are uniformly distributed within the λ_1 intervals, which is reasonable in the absence of any additional information.

0.67. We should thus prefer to send a rather than b to λ_2, since it is more likely to be decided there.

We are now able to handle tuples individually, by exploiting the information supplied by version λ_1, namely the probabilities $P_{i+1,i}(t), P_{i+2,i}(t), \ldots, P_{n,i}(t)$.

$M_{j,i}(t) = 1 - P_{j,i}(t)$ is the probability that t will remain MAYBE at λ_j if it is sent there from λ_i. But, we already had a generalized estimate of this probability, which we could obtain if we did not differentiate between tuples: $\frac{m_j}{m_i}$, because m_j tuples remain MAYBE at λ_j and m_i of them are MAYBE at λ_i. We were thus able to avoid using the *average* behavior over all tuples and we could derive an estimate of the probability that an object will remain MAYBE for each individual tuple.

Example 2. Consider a, b with $P_{2,1}(a) = 0.75, P_{3,1}(a) = 0.99$ and $P_{2,1}(b) = 0.25, P_{3,1}(b) = 0.99$. If we did not differentiate between a and b, then we would use their average behavior, namely $m_2 = 0.5, m_3 = 0.01$.[2] If $c_2 = 1, c_3 = 2$, then the optimal generalized plan would be λ_1, λ_2 with a cost of $1 + 0.5 \cdot 2 = 2$. But, if we handled tuples individually, we would decide that for a the optimal plan is λ_1, λ_2 with a cost of $1 + 0.25 \cdot 2 = 1.5$, while for b the optimal plan is λ_2 with a cost of 2, for a per-tuple cost of 1.75.

For the example above, we were able to reduce the cost of the OGP which was 2 to and individualized cost of 1.75. Incidentally, IIP has a cost of $0.5 + 0.49 \cdot 2 = 1.48$. So, utilizing the resolution probability has helped us cover some ground between OGP and IIP.

A simplified block diagram of individualized tuple handling is shown in Figure 5. Rather than sending a MAYBE tuple to the immediately next version of the train (as in Figure 1), OGP is invoked and picks the next version. The algorithm is listed fully in Figure 6.

4.2 Benefit of Individualization

Individualized handling is possible only when different tuples have different resolution probabilities; otherwise, the individual decision on how to handle each tuple will coincide with the generalized decision. The limit case when resolution probabilities are maximally biased, i.e., close to either 1 or 0 produces the maximum benefit; in that case, knowledge about where each tuple will be resolved approaches the omniscience of IIP.

The practicality of individualization depends therefore on at least two factors: (i) the ability to compute resolution probabilities, and (ii) their sufficient "spread" around the average behavior to make individualization attractive.

A third factor should also be considered, namely per-tuple costs. If we re-examine Example 2, but with $c_2 = 1, c_2 = 10$ we will come to different conclusions. Now, OGP will have a cost of $1 + 0.5 \cdot 10 = 6$ and IIP will have a cost of $0.5 + 0.49 \cdot 10 = 5.4$. It is apparent that the potential benefit of individualization (fraction of cost of OGP that we may gain) is reduced. Indeed, we can calculate that for both a, b the optimal plan is now λ_2, λ_3, and hence individualized handling will not benefit us at all compared to OGP.

For practical purposes, we suggest that the cost optimizer should first compute C_{IIP} and compare it with C_{OGP}. If it

[2]More precisely $\frac{m_2}{m_1} = 0.5$, and $\frac{m_3}{m_1} = 0.01$, but we can renormalize m_2, m_3 to reflect the fact that we are now deciding how to handle the m_1 fraction of tuples output by λ_1 rather than the $m_0 = 1$ fraction of tuples of the input.

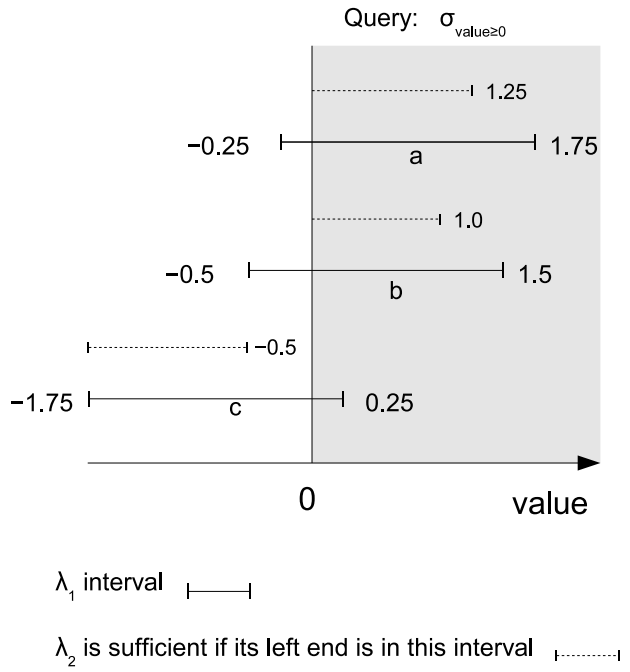

Figure 4: Calculation of $P_{j,i}(t)$

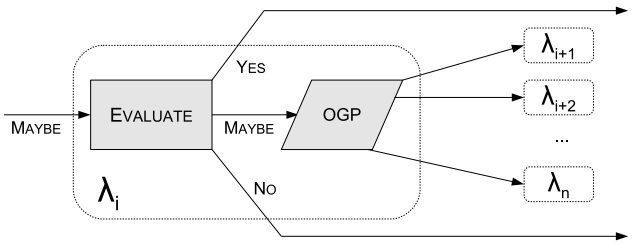

Figure 5: Individualized tuple handling

substantially lower, then the potential benefit of individualization merits consideration.

The actual benefit of individualization cannot be easily predicted and may change over time. If C_{IIP} is substantially lower than C_{OGP} we suggest that a fraction f_i of input tuples should be handled individually by each version, and $1 - f_i$ should be handled using the OGP. The per-tuple cost of the two sets can then be monitored either a priori (by comparing the optimizer's predicted C_{OGP} vs. the average cost over tuples handled individually) or a posteriori (by comparing the actual per-tuple cost for tuples handled by OGP or individually). Whenever C_{OGP} performs significantly worse, f_i could be increased, with more tuples handled individually, and vice versa.

Of course, in principle, C_{OGP} will always perform worse than individualized tuple handling. However, we note that individualized tuple handling involves an overhead proportional to f_i, since resolution probabilities must be calculated, and the optimizer must be called for each tuple handled individually. In our example, if $c_2 = 1, c_3 = 2$ then we gain 0.25 by individualizing the handling of a, b but pay the cost of

```
(1) class Version {
(2)   integer i; // this version is λ_i
(3)   integer n;
(4)   Version v[1..n];
(5)   real c[1..n]; // per-tuple cost
(6)   stream of tuple out; // Output of MVP
(7)
(8)   function EVALUATE // Determines status of a tuple
(9)   INPUT: tuple t;
(10)  OUTPUT: status ∈ {YES, NO, MAYBE};
(11)
(12)  function RESPROB // Estimate P_{j,i}(t)
(13)  INPUT: tuple t; integer j;
(14)  OUTPUT: real P;
(15)
(16)  procedure TUPLEHANDLER // Handle one tuple
(17)   INPUT: tuple t;
(18)
(19)   status ← EVALUATE(t);
(20)   if (status=YES or
(21)      (status=MAYBE and i=n))
(22)        out.add(t);
(23)   else if (status=MAYBE)
(24)      if (i=n−1)
(25)        v[n].TUPLEHANDLER(t);
(26)      else {
(27)        real P[i+1..n];
(28)        for j ← i+1 to n
(29)           P[j] ← RESPROB(t, j)
(30)        real cost; set of integer included;
(31)        (cost, included) ← OGP(1−P, c);
(32)        integer next ← min(included);
(33)        v[next].TUPLEHANDLER(t);
(34)      }
(35) }
```

Figure 6: Individualized tuple handling algorithm at version λ_i

computing resolution probabilities and computing optimal plans for the two tuples. Moreover, even if individualization cost is negligible, then it may still be counterproductive to choose it, since the resolution probabilities may be inaccurate, hence leading in worse observed performance.

4.3 Tuple Grouping

To estimate $P_{j,i}(t)$ in the case of intervals, we relied on domain knowledge which allowed their direct (theoretical) computation. However, in general, this will not be feasible for two reasons: (i) it will be difficult to compute from first principles the probability that e.g., a face will be detected by `FaceDetector` given the output of `ObjectDetector`: creating a formal probability model would be difficult and require looking into the innards of the different expensive methods; (ii) even if such a model was possible, it might be expensive to compute, unlike the case of our interval examples, where the resolution probabilities for a tuple output by version λ_k can be evaluated in $O(n-k)$ time.

Tuple grouping is a different adaptive strategy with intermediate granularity between OGP and Individualized handling. The key observation is that the benefit of individualization is not so much the result of our knowledge of resolution probabilities, but rather due to the fact that handling different subsets of tuples separately may result in a reduction of the overall evaluation cost.

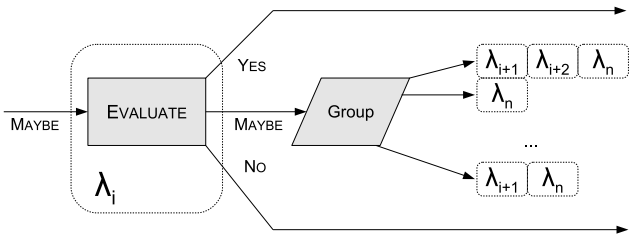

Figure 7: Tuple grouping

For example, consider a system for identifying certain "objects of interest" in a video stream, similar to the example given in the Introduction. The `ObjectDetector` method may work by computing the difference between a reference "empty" frame and the frame captured from the camera. In addition to the binary decision of whether the frame contains an object or not, it could output an additional piece of information which is computed anyway during query evaluation, namely the `size` of the detected foreign object, quantified, e.g., by the difference between the reference and captured frame. Since different objects of interest may have a different size (e.g., people vs. vehicles), knowledge of the `size` variable conveys some information about the probability that different objects would be decided by subsequent versions. In our example, very large objects would probably be rejected by the `FaceDetector` while face-size ones would have a higher probability of being accepted. The opposite would be the case if cars were our objects of interest. Similarly, a `height` attribute would also correlate with the decision of `FaceDetector`, as human faces would tend to occupy a vertically intermediate position within the observed space.

We do not need to model $Pr(face|size)$ explicitly, e.g., by considering distance, position of the camera, etc., in an elaborate model. Rather, we can group objects based on the `size` variable, into classes of varying size, and then treat the resulting tuple subsets separately. Whereas OGP relies on statistics about how many objects are MAYBE at each version, and Individualized handling computes the chance that an individual tuple will remain MAYBE, this "tuple grouping" approach requires only statistics about how many objects are MAYBE at each version, conditioned on the tuples' subset membership based on the informative variable.

A block diagram of Tuple Grouping is shown in Figure 7. The grouping operator intercepts the MAYBE stream of a version and splits it into different trains which are evaluated separately. Identifying which attributes to use for grouping is an interesting challenge for future work. We suggest that MAYBE tuples should be padded with extra attributes that each version can expose to the query processor; additionally, such tuples can carry along their processing cost by summing the costs of evaluated versions. At the output interesting attributes can be identified by identifying low-entropy (uneven) distributions of cost over candidate attributes.

5. EXPERIMENTS

In this section we study experimentally the techniques presented in our paper. To investigate a large parameter

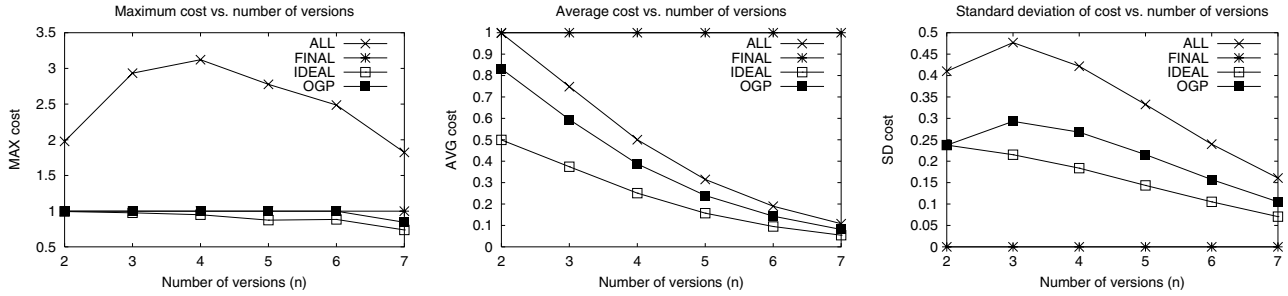

Figure 8: Maximum, average, and standard deviation of cost for ALL, FINAL, IDEAL, and OGP

space we carried out most of our experiments using R [10] and counting the costs of expensive predicates, rather than waiting for them; we have also tested the relative performance of different plans using our Java-based prototype on a Pentium 4/2.8GHz PC with 512MB, with Windows XP.

5.1 Relative Performance

First, we compare the cost of ALL, FINAL, IIP, and OGP. The specific performance differences between these depend on the per-tuple cost, M-selectivities, and the number of different versions. We compare them across a wide range of these parameters to illustrate their behavior.

For a given number of versions n, we set $c_n = 1$ and then choose $c_i \sim U(0, c_{i+1})$ where $U(a, b)$ refers to a uniform distribution in interval (a, b). This choice ensures that versions have increasing cost, as required, but makes the maximum entropy (most random) assumption about their relative costs. Similarly, we compute M-selectivities by setting $m_1 \sim U(0, 1)$ and $m_i \sim U(0, m_{i-1})$. The number of versions n is varied in the set $\{2, 3, \ldots, 7\}$.

For each choice of n we generate 10,000 sequences; we observed little difference in our results compared to 1,000 sequences, assuring us that 10,000 sequences are sufficient for our estimates to converge. For each plan, we measure four metrics: MIN, MAX are the minimum/maximum cost over all 10,000 sequences, AVG is the average cost, and SD is the standard deviation of the cost.

MIN was close to zero for all plans except FINAL which always pays the cost of $c_n = 1$; this is because over the 10,000 randomly generated sequences it is very likely that at least one will have c_1, m_1 close to zero. Thus, we only plot MAX, AVG, SD with varying n in Figure 8.

Note that ALL has occasionally very poor performance as measured by MAX; this is because when cheaper versions do not decide many objects, then their evaluation is an overhead paid needlessly by ALL. By contrast, FINAL evaluates only λ_n, thus the worst case performance of OGP is upper bounded by that of FINAL. OGP occasionally decides to use only λ_n and exploits additional versions if it can.

This robustness of OGP is also evident by SD. The cost of OGP is less variable than that of ALL. Moreover, OGP outperforms ALL on average. These observations motivate us to use OGP since it allows us to combine the benefits of FINAL and ALL: like FINAL it has a good worst-case performance, never paying more cost than c_n irrespective of the characteristics of the input data; like ALL it can use the auxiliary versions to reduce the cost to a fraction of that produced by FINAL (on average).

5.2 Prototype Performance

We tested OGP, ALL, and FINAL on our Java-based prototype. The basic architecture is to assign a thread and a queue to each version; each version pushes tuples to the queue of more expensive versions when they are MAYBE. The cost of a predicate is simulated by busy wait of prescribed time. We set $c_n = 10ms$ and decided the costs and M-selectivities of the other versions as previously described. We varied $n \in \{2, 3, \ldots, 7\}$. We carried out 50 experiments for each choice of n, each of them over a 1,000-tuple input. To conserve space we report the summary of our results over the entire parameter space in Figure 9. The results confirm our observations from the previous subsection about the robustness and good performance of OGP.

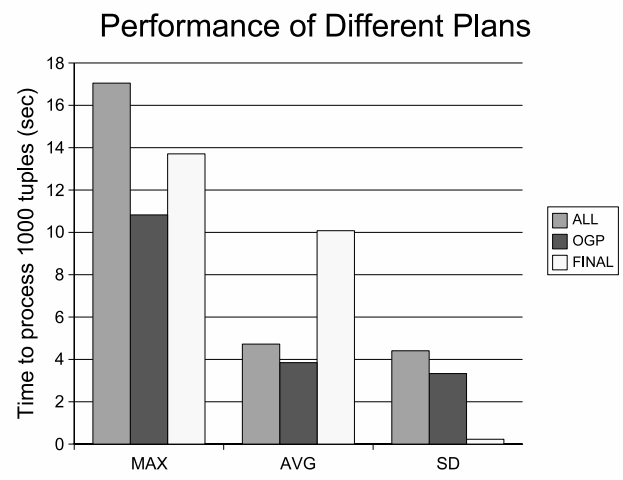

Figure 9: Performance of different plans in MVP prototype system

5.3 Shared Cost

We now study how shared costs affect the evaluation cost. As previously explained, if λ_k is evaluated then we can occasionally use part of its cost for more expensive versions, subtracting it from their own cost. In this experiment, we followed the same procedure to generate our test data including "full" per-tuple costs, which assume that tuples arrive directly from the input and thus no cost has been saved. Then, for each version λ_k with $k \in \{2, 3, \ldots, n\}$ we compute conditional costs as cost[k][l] = cost[k][0]$-s\cdot$cost[l][0]. In other words, we assume that a fraction of shared cost s

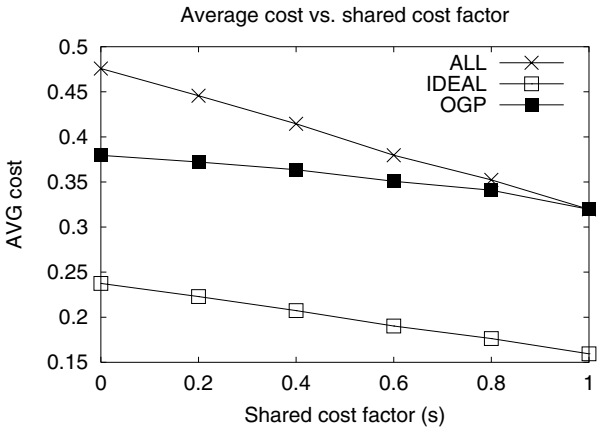

Figure 10: Average cost vs. fraction of shared cost

of the full cost of λ_l can be subtracted from the full cost of λ_k. We vary $s \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ and n as previously.

We notice (in Figure 10) that as the shared cost increases, the average per-tuple cost decreases; this is expected, since a greater fraction of the cost of cheaper versions can be subtracted from the cost of the more expensive ones. Interestingly, the worst performance of ALL is when there is no shared cost, while for $s = 1$ ALL and OGP become identical. If all the cost paid by cheaper versions can be subtracted from the more expensive ones, then the optimal decision is to always evaluate all the versions: if they decide a tuple, then we avoid evaluating the more expensive versions; if they do not, then our effort was not useless but can be written off from the more expensive versions.

5.4 Selectivity Estimation Errors with Sampling

The good performance of OGP depends on the possession of accurate estimates of M-selectivities. These can be estimated from the input data by sampling a fraction f of the input tuples and applying the ALL plan to them, noting what fraction of tuples f_k remains MAYBE in each version λ_k. Then, we can estimate $m_k = \frac{f_k}{f}$.

There is an interesting tradeoff between f and the evaluation cost: as f increases, selectivity estimates improve, as they are obtained from a bigger sample. At the same time, the cost of selectivity estimation increases as a greater fraction of tuples are handled using ALL which is not optimal.

To study this tradeoff, we conducted an experiment varying: (i) N is the number of tuples in the input and varies in $\{10^3, 10^4, 10^5, 10^6\}$, (ii) n as previously, and, (iii) the sampling ratio $f \in \{0.01, 0.025, 0.05, 0.1, 0.15, 0.2\}$.

We plot the results in Figure 11. The observed cost is the sum of the cost paid in the f fraction of tuples that are sampled and handled by ALL plus the cost paid in the remaining $1 - f$ fraction handled by OGP using the estimated M-selectivities. We divide this by the cost of OGP assuming that the true M-selectivities were a priori known.

We observe that the observed cost improves as the number of tuples N increases, because we are now sampling more tuples and are able to approximate the M-selectivities better. As the number of versions n increases, performance deteriorates since it becomes increasingly unlikely that the correct plan will be discovered by OGP (the search space increases). Finally, as f increases we observe an initial improvement as a bigger sample allows more accurate selectivity estimation; however, further increase of f leads to deterioration, as more tuples are now handled by the relatively inefficient ALL plan.

5.5 Individualized Tuple Handling

We now study the effect of individualized tuple handling on the evaluation cost. Individualized tuple handling provides a benefit when tuples have different probabilities of being resolved in the different versions, and especially when these tend to be close to 1 and 0.

In this experiment we generate per-tuple costs as before, but for each run we generate batches of 1,000 tuples each of which has a different set of resolution probabilities. We choose for each batch $M_{1,0}(t) \sim \mathcal{B}(a, a)$ and we set for $i \in \{2, \ldots, n\}$ that $M_{i,0}(t) \sim M_{i-1,0}(t)\mathcal{B}(a, a)$.We use the *Beta* distribution to produce a probability in the $(0, 1)$ interval that can either have a mode in 0.5 (less differentiation) or a U-shape with more tuples towards 0 and 1 (more differentiation). In particular, if $a = 1$ resolution probabilities are uniformly distributed, if $a < 1$ they are biased to the extremes, and if $a > 1$ they are more uniform; we will call a the *tuple conformism*. We vary $a \in \{0.1, 0.5, 1, 5, 10\}$ and show the results in Figure 12.

We note that as tuple conformism increases, the Individualized plan approaches OGP. This is expected, since the advantages of individualization are eroded when tuples exhibit a more average behavior. By contrast, as tuple conformism decreases, the Individualized way approaches the IIP plan. It should be noted that this IIP can never be attained even if resolution probabilities were even closer to 0 and 1, because the IIP's omniscience allows it to "cheat" by not evaluating λ_1 for $1-m_1$ tuples, and not evaluating λ_n for the m_n tuples that are never resolved.

5.6 Tuple Grouping

In this experiment we assess the efficacy of tuple grouping and demonstrate how it approaches the Individualized and OGP limits as the quality of the grouping attribute increases/decreases. We vary n as before. For each n we generate 10,000 multi-version predicates. For each of these, we generate version costs randomly as previously, and vary the number of groups $n_g \in \{2, 3, \ldots, 7\}$ as well as a grouping attribute imprecision factor $noise \in \{0.1, 0.2, \ldots, 0.5\}$.

Our approach is the following: the grouping attribute is meant to distinguish between tuples that require different costs to be decided. Thus we seed the grouping attribute with the value of the individualized handling cost of each tuple—computed as in the previous section and with $a = 0.1$, since we are interested in the case where Individualized handling makes a big difference compared to OGP. Subsequently, we add normally distributed noise to the grouping attribute with mean 0 and standard deviation *noise*. This reduces the quality of the attribute by various amounts. Finally, we scale the grouping attribute to the $[0, 1]$ interval and divide it into n_g buckets of equal width. Thus, tuples are split into groups defined by the bucket id, and each group is optimized separately using the OGP.

Results are shown in Figure 13. We observe that as *noise* increases, the performance of the tuple grouping strategy deteriorates from an initial level close to Individualized to a level close to OGP. Thus, tuple grouping can be attrac-

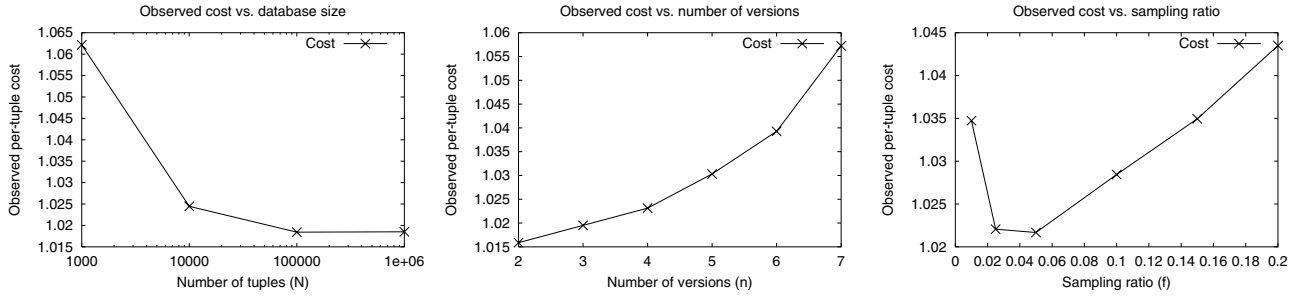

Figure 11: Per-tuple cost as a function of input size, number of versions, and sampling ratio

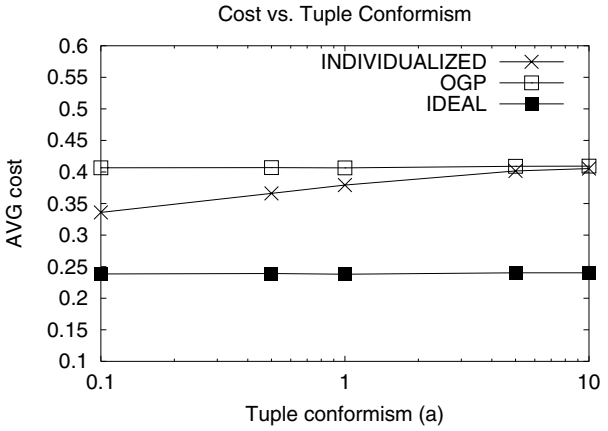

Figure 12: Average cost vs. tuple conformism

tive to reduce the cost of an MVP below that of OGP, as long as grouping attribute(s) which are predictive of evaluation cost can be identified. If the grouping attribute is completely noisy then there is no correlation between it and the evaluation cost of tuples: thus, the different tuple sets will be similar in terms of costs, and the grouping strategy is reduced to OGP with a slight overhead. Hence, we recommend that if an attribute correlates with evaluation cost, grouping based on that attribute should be attempted.

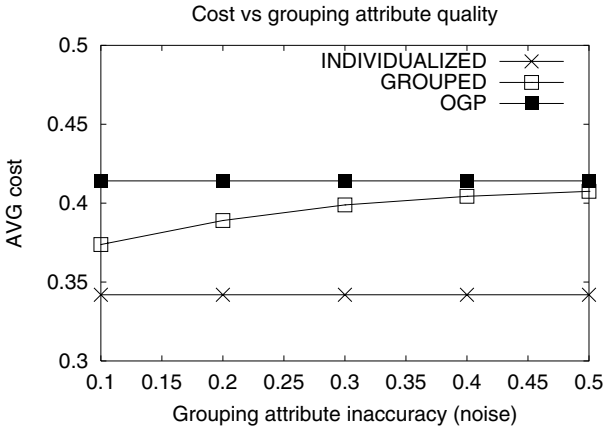

Figure 13: Average cost vs. quality of tuple grouping

5.7 Semi-Honest Versions

Our assumption of version honesty may only be approximately correct in practice. If a cheap predicate is not completely honest then this may introduce false positives into the answer, since objects that were output as YES may really be classified as NO or MAYBE by the final version. False negatives may also be introduced, because objects that were rejected as NO may really be YES or MAYBE.

To study the effect of semi-honest versions, we carried the following experiment. We simulated a Wiener process [13] which stands for the temporal fluctuations of a monitored attribute. Instead of approximating it with an interval, we decided to transmit its value periodically to a number $n \in \{2, 3, \ldots, 7\}$ of different recipients. There are thus n different versions of the value, including the sensor recording the attribute (λ_n). The i^{th} version corresponds to a replication period of 2^{n-i}. Thus, the value according to λ_i is normally distributed with variance 2^{n-i}.

We chose the sensor values randomly $\sim U(-100, 100)$, and then generated the additional versions according to the above procedure. The query was set to ask for values ≥ 0. A version decided that its value was YES if there was more than $1 - conf$ probability of being above 0, and NO if this probability was less than *conf*. The parameter *conf* was varied in $\{0.01, 0.05, 0.1, 0.2, 0.4\}$. Our experiments were repeated 10,000 times for each choice of n, and for each of these runs we generated costs randomly 100 times as previously described. Furthermore, we varied *conf* and measured the cost of evaluating the query as well as the sum of the false positives and false negatives normalized by dividing with the number of input tuples.[3]

The results are shown in Figures 14, 15. As *conf* increases, versions are more aggressive about deciding objects as YES or NO. The effect of this is that the cost of the execution is reduced, since more objects exit the train of versions earlier; however, the aggressiveness has the negative effect of increasing the number of false positives and negatives, as decisions taken with less confidence have a greater chance of backfiring.

5.8 Side Benefits of Predicate Optimization

As hinted in the Introduction, optimization of MVPs is not just the means to reduce the cost of a single component of a more complex query, but can have additional side benefits. Take any execution plan, maintain its structure, but replace the standard predicates with MVP-optimized ones:

[3]Due to the symmetry of the simulation, the numbers of f.p.'s and f.n.'s were approximately equal.

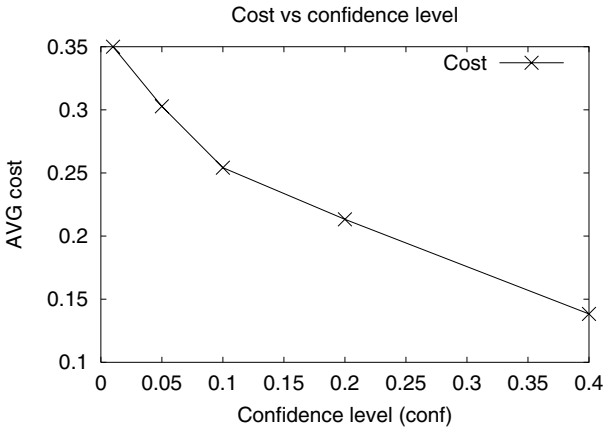

Figure 14: Effect of decision aggressiveness on cost for semi-honest versions

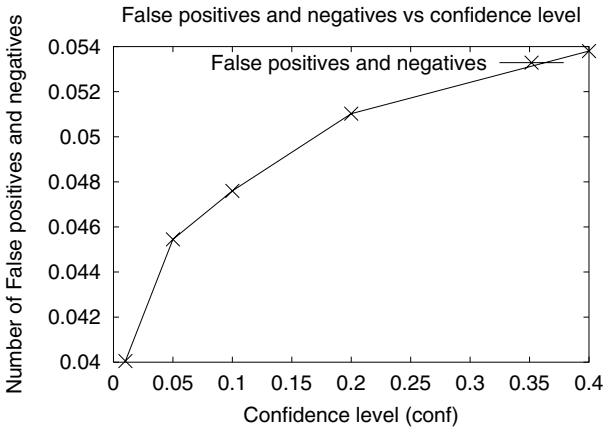

Figure 15: Effect of decision aggressiveness on the number of false positives/negatives for semi-honest versions

if the input cardinality of an MVP is N and $\frac{C_{OGP}(S)}{C_{\text{ALL}}(\lambda_n)} = b$, then the cost will be reduced by amount $N(1-b)C_{\text{ALL}}(\lambda_n)$. However, by reducing the cost of the MVP, it is possible that the query optimizer will choose a different arrangement of operators, which will result in even lower cost.

To test this intuition, we generated a series of n_p predicates, varying $n_p \in \{2, 3, \ldots, 7\}$. Each predicate has a random per-tuple cost $c(i) \sim U(0,1)$ and a random selectivity $s(i) \sim U(0,1)$. We generated 10,000 such series for each choice of n_p. For each series, we first calculate the optimal cost $CFINAL$ of evaluating their conjunction by optimally ordering them in increasing $\frac{c(i)}{1-s(i)}$. Then, we vary a reduction factor $rf \in \{0.1, 0.2, \ldots, 0.9\}$ and reduce the cost of the last predicate in the optimal order by amount rf.[4] We then compute the cost $COGP$ which uses the reduced cost but in the same order as $CFINAL$. Finally, we compute $CADJUSTED$ which uses the reduced cost but with the new optimal order.

Results are shown in Figure 16; in each plotted point we average over all series choices for n_p. If the optimized predicate is left in its original position ($COGP$), then there is a performance benefit which increases as the amount of reduction in its cost (rf) increases. However, even greater savings are effected by the optimizer by re-arranging the predicates ($CADJUSTED$).

[4]If this already has cost $< rf$ we do not change its cost.

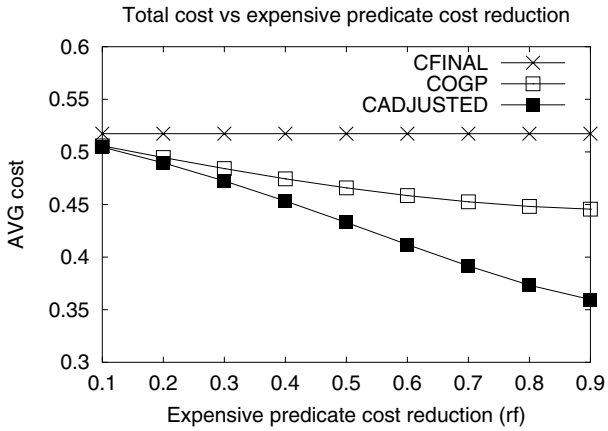

Figure 16: Benefit of optimizing a single MVP on a multi-predicate query

We are currently investigating the effects of MVP optimization in general query plans. An additional source of cost reduction not investigated here arises from the fact that the versions of the MVP can be "split" by placing other operators between them. This extra degree of freedom in plan choice can result in even greater cost reductions.

5.9 Cost of OGP

The OGP optimizer forms the basis of our techniques, so we measured its cost using a Java implementation. Averaged over 10^6 runs and for $n = 2$ to 7, its cost ranges from 0.36μsec to 1.98μsec. OGP is most heavily used in the individualized plan, where, to handle a tuple, it is invoked as many as $n-2$ times in the worst case. Thus, even for $n = 7$, the total cost of OGP per tuple does not exceed $10^{-5}s$ in our modest machine, which is low compared to the anticipated running time of expensive predicates.

6. RELATED WORK

Optimizing queries with expensive predicates has been studied in the database literature, e.g., by Hellerstein and Stonebraker [7], Kemper et al. [8], Chaudhuri and Shim [5], Babu et al. [2]. Our work is orthogonal to theirs, since our goal is not to show how to order predicates or how to place them between other relational operators; rather it is to reduce the per-tuple cost of the predicates themselves by making them adaptive to the characteristics of the input and either adding or removing cheap predicates and routing tuples between them intelligently. As previously mentioned, optimizing expensive predicates may have side benefits above and beyond the cost reduction of the expensive methods.

Lazaridis and Mehrotra [9] presented a scheme for approximate query processing with set-based results which used the concept of a "probe" of an imprecise object to retrieve its precise version. The idea of saving the cost of the probes by exploiting less accurate objects is similar to our idea of placing multiple cheap versions in front of the expensive ones.

Our work differs since we consider multiple levels of cheaper versions, and we show how to organize them, and route tuple between them, which was not an issue with that work, as there were only two versions to deal with. The idea of applying a cheap test before a more expensive one is found also in spatial databases [4] where spatial access methods (SAMs) can be used to generate candidates in a *filter* step and the exact answer is produced by examining candidates' exact geometry in a subsequent *refine* step.

The idea of tuple grouping is similar to that of content-based routing (CBR) suggested by Bizarro et al. [3]. More specifically, to identify candidate attributes for grouping, it is possible to use information gain; the alternative, proposed in this paper is to monitor the cost of different subsets of tuples in the output, in order to identify attributes which exhibit unusual cost. To estimate selectivities we do not need to send a fraction of tuples to each version directly as proposed in the above paper, but can rather route them through the ALL plan. Finally, we suggest that not only the content of tuples but also extra attributes calculated by the operators (e.g., the `size` attribute) may be used for grouping. Using such attributes allows us to exploit the work done by operators without paying the great routing overhead associated with long classifier attributes [3].

Some of the work presented in this paper is similar in flavor to the idea of Eddies proposed by Avnur and Hellerstein [1]. The Optimal Generalized Plan is more traditional than Eddies, since it routes tuples in a fixed (generalized) order, although it goes beyond traditional query optimization by choosing whether or not to include some operators rather than just picking physical implementations of the operators and arraning them in a manner that reduces the cost. Individualized handling is more eddy-like, while tuple grouping is intermediate, splitting tuple streams into multiple separately optimized groups of tuples. Note that our algorithms can co-exist with either a traditional System R [11] type of architecture or with an eddy-based architecture. From the point of view of the former, our techniques can be viewed as simply introducing a new more efficient algorithm for evaluating a selection condition; in terms of the latter, our techniques can be implemented within the eddy as a way of routing tuples among a set of operators having a specific property (honesty), in which the set of operators to be evaluated is adaptive, while their order is not.

7. CONCLUSIONS AND FUTURE WORK

We introduced the concept of multi-version predicates in order to reduce the cost of expensive user-defined functions in a database system. Direct optimization of such functions reduces their runtime evaluation cost, and may help the query optimizer, by making them more attractive for early evaluation. Our main idea is to use pre-filtering predicates that resolve part of the input at lower cost. We also suggested two ways in which additional predicates can reduce the cost: by exploiting shared costs, and by exporting information that can be used for tuple grouping and separate optimization. We have identified the optimal fixed-order handling scheme (OGP), and showed that individual-tuple optimization can also be performed using the OGP procedure. We proposed tuple grouping for optimization at an intermediate granularity. If tuples deviate sufficiently from the average stream behavior, then our adaptive techniques may reduce the cost below what is possible with OGP.

We are pursuing two main directions of future work. First, we want to examine how multi-version predicates fit within the query optimizer. In particular, we want to quantify their effect on general query plans, and to pursue a further avenue of optimization, namely MVP-splitting, i.e., allowing the versions of an MVP to be interspersed with other operators. Second, we are studying how extra pieces of information—beyond costs and cardinalities—such as exported attributes, shared costs, or resolution probabilities, can be exploited in query optimization.

8. ACKNOWLEDGMENTS

This work was supported by the NSF under Award Numbers 0331707 and 0331690. We thank Chen Li and Nikil Dutt for their comments during a presentation of an earlier version of this work, as well as the anonymous reviewers for their useful suggestions.

9. REFERENCES

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of SIGMOD Conference*, 2000.

[2] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proc. of SIGMOD Conference*, 2004.

[3] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing: different plans for different data. In *Proc. of VLDB Conference*, 2005.

[4] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step Processing of Spatial Joins In *Proc. of SIGMOD Conference*, 1994.

[5] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

[6] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst.*, 23(2):113–157, 1998.

[7] J. M. Hellerstein and M. Stonebraker. Predicate migration: optimizing queries with expensive predicates. In *Proc. of SIGMOD Conference*, 1993.

[8] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proc. of SIGMOD Conference*, 1994.

[9] I. Lazaridis and S. Mehrotra. Approximate selection queries over imprecise data. In *Proc. of ICDE Conference*, 2004.

[10] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[11] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of SIGMOD Conference*, 1979.

[12] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proc. of VLDB Conference*, 2006.

[13] E. W. Weisstein. Wiener process. *From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/WienerProcess.html.*