Preference Query Evaluation Over Expensive Attributes*

Justin J. Levandoski Mohamed F. Mokbel Mohamed E. Khalefa
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA
{justin,mokbel,khalefa}@cs.umn.edu

ABSTRACT

Most database systems allow query processing over attributes that are derived at query runtime (e.g., user-defined functions and remote data calls to web services), making them *expensive* to compute relative to relational data stored in a heap or index. In addition, core support for efficient *preference query processing* has become an important objective in database systems. This paper addresses an important problem at the intersection of these two query processing objectives: efficient preference query evaluation involving expensive attributes. We explore an efficient framework for processing *skyline* and *multi-objective* queries in a database when the data involves a mix of "cheap" and "expensive" attributes. Our solution involves a three-phase approach that evaluates a correct final preference answer while aiming to *minimizing* the number of expensive attributes computations. Unlike previous works for distributed preference algorithms that assume sorted access over each attribute, our framework assumes expensive attribute requests are *stateless*, i.e., know nothing previous requests. Thus, the proposed approach is more in line with realistic system architectures. Our framework is implemented inside the query processor of PostgreSQL, and evaluated over both synthetic and real data sets involving computation of expensive attributes over real web-service data (e.g., Microsoft MapPoint).

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

Embedding preference query evaluation in database systems has proven to be very important, having spawned the realization of useful, non-trivial systems such as context and preference-aware databases [16, 17, 23] that in turn enable exciting new application scenarios. Preference queries return to a user a set of *preferred answers* from a large set of multi-dimensional objects. An example of a preference query is: "*find my preferred restaurants based on my preferences for price, rating, driving time, and current wait time*". Here, each preference represents a "dimension" of a restaurant object. The answer to such a query depends on the preference method used to evaluate the query. Two of the most popular preference evaluation methods are the skyline [4], and its derivative the multi-objective method [1]. For our example query, a skyline would return the set of restaurants that are not dominated by other restaurants. A restaurant x dominates another restaurant y if x is better than y in one dimension (e.g., price, rating, driving time, or wait time), while it is better-than or equal to y in all other dimensions. The multi-objective preference method, meanwhile, *combines* some dimensions using a monotone scoring function, and performs a skyline over the newly transformed objects. For our example query, the multi-objective method may sum the *drive time* and *wait time* attributes into a single dimension and perform the skyline over three dimensions: price, rating, and *total* time.

An important systems issue behind implementing preference query processing in database systems is gracefully handling *expensive* attributes. There are two main sources of *expensive* attributes that are widely supported in most database systems, commercial or otherwise: (1) *User-defined functions* (abbr. UDFs). Attributes retrieved through UDFs are considered *expensive* if the function incurs high computational overhead (e.g., distance computation in a road network). (2) *Third-party retrieval.* Attributes retrieved through third-party data sources are considered *expensive* due to the network overhead/delay in transmitting the values of this attribute to the local table (e.g., restaurant rating stored at a third party website). Coupling preference query processing with a mix of "cheap" and "expensive" attributes changes the algorithmic cost model. Computation over cheap attributes are essentially *free* relative to computation/transmission of expensive attributes. To test this disparity, we ran a simple experiment comparing retrieval of third-party web data to local disk reads in our system implementation in PostgreSQL (setup in Section 7). The retrieval of a single expensive attribute (driving time from the Microsoft MapPoint web service [19]) takes 502 ms. Alternatively, the time needed to read a cold and hot 8 KB buffer page from disk is 27 ms and 0.0047 ms, respectively.

*This work is supported in part by the National Science Foundation under Grants IIS-0811998, IIS-0811935, CNS-0708604, IIS-0952977 and by a Microsoft Research Gift

CIKM'10, October 26–30, 2010, Toronto, Ontario, Canada.

Clearly, local DBMS operations incur order of magnitudes less cost than retrieving a *single* expensive attribute.

A straightforward solution for skyline and multi-objective preference queries given a mix of expensive and cheap (i.e., local) attributes would first request *all* expensive attributes for *all* objects in the data. Then, any skyline or multi-objective algorithm can be invoked to compute the final answer. This method is prohibitively expensive, as it makes the maximum amount of *unnecessary requests*. As we will show, *not all* expensive attributes need retrieval to compute a correct skyline or multi-objective answer. Thus, development of a new framework that minimizes the *unnecessary requests* for expensive attributes is necessary.

In this paper, we propose an efficient preference evaluation framework for skyline and multi-objective preference queries that involve *expensive* attributes. The efficiency of this framework comes from its ability to prune objects without computing their *expensive* attributes, thereby reducing *unnecessary requests*. Our framework can be summarized by three main phases: *initialization*, *pruning*, and *cleaning*. *Phase I, the initialization phase,* analyzes the "cheap" attributes in order to find a set of initial objects that are *guaranteed* to be in the final preference answer (abbr. S_c). *Phase II, the pruning phase,* utilizes the results obtained from the initial phase to make a range request to retrieve the expensive attributes for a small sample of objects that are *not* in S_c. Phase II then prunes incomplete objects (i.e., objects *without* their expensive attributes) that are guaranteed *not* to be in the final answer regardless of the actual value of their expensive attribute(s). Pruning reduces the number of total *unnecessary requests* in our framework. *Phase III, the cleaning phase,* makes a random-access request to retrieve the expensive attributes for the remaining (non-pruned) objects, and computes a final preference answer.

Our framework can be seamlessly added to existing systems as: (1) We do not have any special requirements for the underlying skyline or multi-objective algorithm used in the first phase, thus, any existing algorithm can be used. (2) We require only *two* fundamental access operations to retrieve expensive attributes: *Random-access* (given an object id, return the attribute value for that object), used in the first and third phases, and *Range-access* (given a certain value v, return object ids and attribute values for the objects whose values below v), used in the second phase. Most importantly, our framework does not hold special assumptions about expensive attribute sources (e.g., third-party, UDFs). We simply assume sources (a) are *stateless*, i.e., do not store information about previous requests, and (b) do not necessarily store data in a sorted order (especially true when asking for a distance/driving time to a given point). These assumptions are in contrast to previous research in our problem space that assume sources store state, and return all attributes in sorted order [2, 18].

The novel contributions of this paper can be summarized as follows. First, we provide an efficient query execution framework for skyline queries involving both *single* and *multiple* expensive attributes. Secondly, we provide an efficient query execution framework for multi-objective queries over both *single* and *multiple* expensive attributes. Third, our methods are implemented *inside* the query processor of PostreSQL [21], and experimentally evaluated using both synthetic and real data sets involving computation of expensive attributes over real web-based data.

The rest of this paper is organized as follows. Section 2 highlights related work. We provide a problem overview in Section 3, and provide an overview of our solution in Section 4. Section 5 presents and efficient preference evaluation framework for skyline evaluation involving both single and multiple expensive attributes. Section 6 discusses our framework applied to multi-objective preference evaluation. Experimental evidence for our framework is provided in Section 7. Finally, Section 8 concludes this paper.

2. RELATED WORK

Processing *expensive*, computationally-bound attributes is a necessity for database systems. The problem has been studied in core database systems [12, 15] for boolean predicates involving computationally-bound attributes. Likewise, data-aggregation system architectures process data from computationally expensive *remote* sources [14]. In this vein, many query processing algorithms have been proposed to deal with remote data ranging from whole query optimization strategies [25] to specific query operators [24]. Recently, preference query processing over computationally-bound attributes has been studied for the case of *top-k* queries [5, 6]. The data access model in these works is similar to ours, where data may come from costly web-based sources [5], or be probed from expensive predicate functions [6]. This is different from our work where we mainly study the problem in the context of "Pareto-optimal" preference evaluation for skyline and multi-objective queries.

Most of existing work in skyline and multi-objective query processing (e.g., see [4, 20]) assume all data is available locally at the same "cost". Exceptions lie in three different categories: (a) skyline query processing over distributed data sources [2, 3, 13, 26]. The distributed skyline web query [2, 3] studies the case where *all* attributes are aggregated from remote sources. However, these algorithms assume that *each* source returns data in *sorted order*. Such an assumption is too strict, and does not follow realistic system behavior (e.g., attribute retrieval over web-services that do not support sorted access [19, 27]). Conversely, in our framework, we do not assume or require sorted access from any data source. (b) peer-to-peer mobile skylines [13, 26] study the case where data is *horizontally partitioned* in a peer-to-peer or mobile network. Conversely, we assume the data is "vertically-partitioned" between relatively "cheap" data attributes and "expensive" computationally-bound attributes. (c) skyline query processing over a mix of non-spatial (cheap) and spatial (expensive) attributes [7, 11, 22]. However, these frameworks assume the computationally-bound attribute always involves a distance computation from a set of data objects to query objects. Thus, optimizations exist for the special case of distance computation in Euclidean or Metric space. We study the more general case of skyline and multi-objective query processing over expensive attributes from *any* domain, rendering spatial optimizations useless.

3. PROBLEM OVERVIEW

System Architecture Assumptions. Our framework supports both the skyline [4] and multi-objective [1] preference methods. Throughout the rest of this paper, without loss of generality, we assume: (1) *Minimum* is better in each dimension, i.e., a user prefers the minimum value over each attribute. (2) *All* relevant attributes (i.e., the attributes

ID	Price	Rating	Wtime	DTime
a	84	78	39	79
b	91	33	19	76
c	27	47	55	62
d	36	95	51	91
e	63	14	39	60
f	1	13	24	80
g	15	12	40	10
h	99	51	30	83
i	35	28	29	9
j	49	29	97	77

Figure 1: Running Example Data

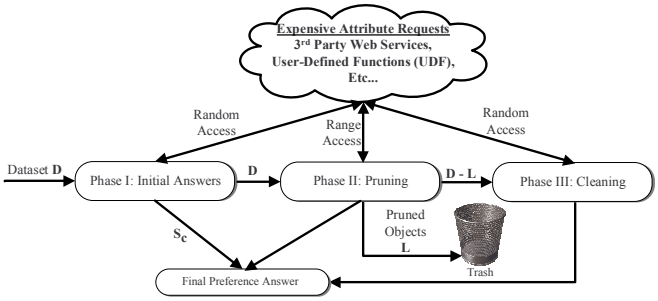


Figure 2: Solution overview

used to compute the skyline or multi-objective answer) must be returned to the user. Also, we assume a realistic system architecture where expensive attribute retrieval *only* relies on (1) *Random-access requests*: given an object *id*, return the attribute value for that object, and (2) *Range-access requests*: given an upper-bound U, return the object ids and attribute values for the objects whose values are less than U. Moreover, we assume the entity that calculates and returns expensive attributes (e.g., web-based data source) is *stateless*, i.e., does *not* store information about previous requests. This assumption contrasts previous attempts to build efficient skyline algorithms over remote, distributed web sources that assume only *sorted* access [2, 18], that require the expensive attribute source to store a *cursor* with the position of *each* sorted access session.

Expensive and Cheap Attributes. Expensive attributes require non-trivial evaluation costs to materialize. For example, (1) the computation of a user-defined function (UDF) or (2) the network overhead induced by requesting/receiving an attribute value from a remote source (e.g., web-based source). We use a simple cost model to denote expensive attributes operations: $cost_c$ is the cost of computing a single expensive attribute while $cost_t$ is the cost to transmit the attribute from the attribute source. On the other hand, cheap attributes require "zero" cost to derive relative to their expensive counterpart. Of course, cheap attributes incur disk I/O costs if they do not reside in memory.

Primary objective. The objective of our framework is to minimize the number of *unnecessary requests* while answering the preference query given a mix of r cheap attributes and s expensive attributes ($r \geq 1$ and $s \geq 1$). We define an *unnecessary request* as the computation and/or transmission of an expensive attribute for a tuple that is *not* in the final preference answer.

Supported Preference Methods. The preference method our framework supports are: (1) *Skyline*. Given a dataset D, a skyline query computes the Pareto-optimal set S of D. S are those objects that are *not dominated* by any other objects. An n-dimensional object x is said to dominate another object y if x is better than or equal to y in n dimensions, and *strictly* better than y in at least one dimension. For our running example in Figure 1, objects $\{b,f,g,i\}$ are the skyline objects. (2) *Multi-objective*. Given a set of n-dimensional objects, multi-objective evaluation combines m dimensions ($1 < m < n$) using a monotone function. A skyline is then computed over the resulting $(n-m)+1$ independent dimensions. For example, in Figure 1, consider a multi-objective query that sums attributes $DTime$ and $WTime$, the answer for this query is $\{f,g,i\}$.

4. SOLUTION OVERVIEW

We now provide an overview of our solution to skyline and multi-objective preference evaluation over expensive attributes. We begin by discussing a naive algorithm and its drawbacks. We then discuss the outline of our solution.

A Naive Solution. A naive solution to our problem of preference evaluation over expensive attributes is as follows. Given N objects where each object has r *cheap* attributes and s *expensive* attributes, request the N expensive attributes from each of the s expensive sources using random access. At this point, all necessary data is materialized and any skyline or multi-objective algorithm can be run locally over the N objects to yield a correct preference answer. This method is exactly how a DBMS would execute the query. The back-end executor will call a given UDF (that contains expensive computations) on every `get_next()` iteration of the scan operator to materialize an attribute.

This naive approach is unnecessarily expensive. If m is the size of the preference answer, this solution makes $s \times (N-m)$ *unnecessary requests* as it requires the retrieval of *every* expensive attribute for *each* object. The cost of this solution is $s \times N \times (cost_t + cost_c)$ where $cost_t$ and $cost_c$ are the costs to transmit the attribute from the attribute source and to compute a single expensive attribute, respectively.

Overview of Our Solution. Figure 2 outlines the main three phases of our framework: the *initialization* phase, *pruning* phase, and *cleaning* phase. Each phase has a *computation* step applied to the "cheap" attributes and a *request* step that issues either a random-access or range request to retrieve "expensive" attributes. It is important to note that these phases are the same for skyline and multi-objective queries for both single and multiple expensive attributes.

Phase I: The *initialization* phase. Given a dataset D, Phase I forms an initial query answer (abbr. S_c) by running the preference query over the "cheap" attributes. A random access request then retrieves the "expensive" attributes for objects in S_c. Phase I does not incur unnecessary requests as it retrieves only the expensive attributes for those objects that are guaranteed to be in the final answer.

Phase II: The *pruning* phase. Given the dataset D from Phase I, this phase performs three main operations: (a) Making a range request to retrieve the expensive attributes for a small sample of objects that are *not* in the initial answer S_c, (b) Creating a pruning set P by combining the returned objects from the range request with some of the objects in S_c. A set of objects $M \subseteq (P - S_c)$ are added to the final preference answer at this point. (c) Using P to prune a set of *incomplete* objects L that are guaranteed not to be in the final answer regardless of their expensive

attribute values. Thus, the efficiency of our framework is to *maximize* the number of objects in L.

Phase III: The *cleaning* phase. This is a final phase takes as input the dataset $(D - L)$, and computes a final answer by first making a random request for remaining incomplete objects in $D - (L \cup S_c)$. Any dominated objects are then discarded. Any remaining objects are added to the final preference answer. Ideally, Phase III is unnecessary as all incomplete (and non-preferred) objects would be pruned by Phase II. Realistically, Phase III may incur some *unnecessary requests*.

5. SKYLINE QUERY PROCESSING

In this section, we present our framework applied to the skyline preference method [4]. We first discuss the query processing case for a single expensive attribute, then discuss multiple expensive attributes.

5.1 Single Expensive Attribute Case

This section discusses a skyline framework involving a single expensive attribute, and one or multiple "cheap" attributes. The overarching goal of this framework aims to minimize *unnecessary requests* for expensive attributes. In this vein, two concepts help our framework achieve this goal: (1) bounding values and (2) U-values. Both of these concepts are used in the *pruning* phase, and will be discussed in detail in subsequent sections. The input to the framework is a data set D, a reference to "cheap" attributes C, and a reference to the expensive attribute source E. Figures 3 will be used to aid presentation by providing a numeric example where attributes *Price*, *Rating*, and *WTime* are "cheap" attributes, and *DTime* is the single expensive attribute. For the rest of this section, the term *e*-values will be used to refer to "expensive" attribute values, while the term *c*-values refers to "cheap" attribute values.

5.1.1 Phase I: Initialization

Phase I of the skyline framework is responsible for (1) finding initial preference answers, S_c, by analyzing only "cheap" attributes, and (2) deriving a *bounding value* v for each object $q \notin S_c$ such that $q's$ *e*-value should be less than v in order to be in consideration for the final answer. Phase I involves three main steps. First, it starts by calculating an initial set of skyline objects, S_c, using only the *cheap* attributes. S_c represents a *subspace* skyline of D over the set of cheap dimensions C. Assuming *no ties* between objects, i.e., objects do not have the same value across all attributes, S_c is guaranteed to be part of the final preference answer. Since each object in S_c is not dominated in subspace C, it cannot be dominated in a larger dimensional space (i.e., over more attributes). Two observations can be made at this point: (1) We do not assume a particular skyline algorithm for computing S_c, thus the storage scheme for the cheap attributes can guide the algorithm choice (e.g., non-indexed [4], pre-sorted [8], or indexed [20]), and (2) We can guarantee that existing skyline algorithms allow us to derive, for each *dominated* object q, a single object p known to dominate it, defined as q's *dominating object*. More than one object may dominate q, but we are only guaranteed to find *one* dominating object as most skyline algorithms rely on the property of *transitivity* to run efficiently. Thus, once an object is found to be dominated, it is instantly discarded (i.e., compared to no further objects). During the computation of S_c, we store, for each dominated object $q \notin S_c$, a pointer to its dominating object $p \in S_c$ that is known to dominate q over cheap attributes C. This pointer will be used as the basis for *bounding values* that are used in the pruning step of Phase II.

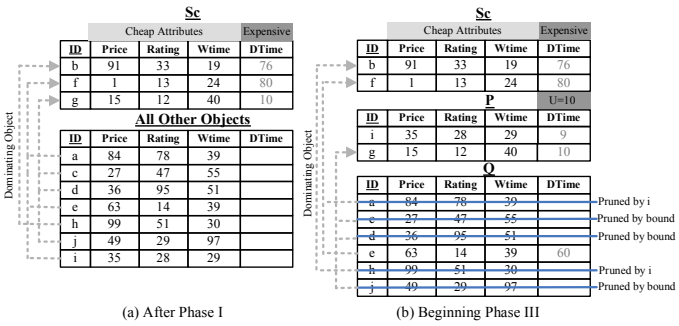


Figure 3: Skyline over a single expensive attribute

Due to efficiency concerns, maintenance of dominating objects requires special attention. For this purpose, we use a *disjoint set* data structure [10]. Each set represents an object p in S_c, along with the objects p is found to dominate during skyline computation. Each set has a representative object (i.e., the *root*), that stores the count of all other objects in the set. In our case, p is the root. We require two efficient operations over disjoint sets. (1) *Set(p,q)*, that sets p to be the dominating object of q. This is an O(1) operation that can be embedded in the dominance check of *any* skyline algorithm that simply stores a pointer from q to p. Later, object f may be found to dominate i, causing set i to point to f, making f the root with a count of three. At this point, f is the dominating object for objects a, i, and c. (2) *Find(q)*, that finds the dominating object for q, an operation important in finding the *bounding value*. *Find* requires traversing the pointer from q to the root of its set. Using disjoint sets, both operations are efficient, and have been shown to exhibit $O(\alpha(n))$ complexity, where $\alpha(n) < 5$ for any practical value of n [10].

Second, Phase I performs its *request step* by retrieving the expensive attributes for *all* objects in S_c. It is important to note that these are necessary requests as objects in S_c are guaranteed to be part of the final answer. Third, and finally, Phase I assigns a bounding value v for each object $q \notin S_c$ as the value of the expensive attribute of q's dominating object. The bounding value represents an *upper bound* of the value of its expensive attribute in order to maintain consideration as a preferred object. Bounding values are used in Phase II for pruning purposes.

Example. Figure 3 depicts the result of Phase I in our running example. The skyline computation using the cheap attributes *price*, *rating*, and *Wtime* results in $S_c = \{b, f, g\}$. Also, there is a pointer from each object $q \notin S_c$ to a correspondingly dominating object $p \in S_c$. For example, object a has a pointer to object f, given f was the object found to dominate a during skyline processing over the cheap attributes. Again, although an object can be dominated by more than one object (e.g., c is dominated by both f and g), we maintain only one dominating object pointer as determined by the underlying skyline algorithm. The values of the expensive attribute values are depicted in gray for objects in S_c. These values also represent bounding values for

the dominated objects not in S_c. For instance, the bounding value for object c is 10 (i.e., its dominating object g's expensive attribute value). Intuitively, if c's expensive attribute is greater than 10, it is guaranteed to be dominated by g.

Cost. The cost of retrieving the expensive attributes in Phase I is influenced by the size of S_c. Let $|S_c|$ be the cardinality of set S_c. Then, Phase I incurs a cost of $|S_c|(cost_c + cost_t)$. In our running example (Figure 3(a)), $|S_c| = 3$, thus the cost is $3(cost_c + cost_t)$. Since object in S_c are guaranteed to be in the final answer, then the number of *unnecessary requests* of Phase I is zero.

5.1.2 Phase II: Pruning

Phase II reduces unnecessary e-value requests by *pruning* some incomplete objects (i.e., objects *not* in S_c and *without* their e-values) that are guaranteed *not* to be final answers.

Phase II has five main steps. First, Phase II starts by calculating a value U that is used in a subsequent range request to the remote source. The U-value has a great effect on the efficiency of our framework, thus options for deriving U will be detailed in Section 5.1.4. Second, Phase II issues a range request to the remote source to retrieve all objects with e-values less than or equal to U. Third, Phase II constructs two sets, the pruning set P that will be used to prune a set of objects without computing their e-values. The set P contains: (1) all objects returned from the range request, denoted P_R and (2) any objects in S_c with e-values less than or equal to U, denoted P_{S_c}, i.e., $P = P_R \cup P_{S_c}$. We also construct a set Q that contains those objects in neither S_c nor P. Two properties hold over objects in Q: (1) their e-values have *not* yet been computed/retrieved and (2) their e-values are known to be dominated by the e-values of objects in P, otherwise they would have been returned by the range request.

Fourth, we clean the set P by removing those objects in P that *cannot* be in the final answer. We do so by running a regular skyline algorithm over the objects in P_R while seeding the initial skyline result with the objects in P_{S_c}. The main idea is that objects in $P_{S_c} \subset S_c$ are guaranteed to be skylines, so, there is no need to run a skyline algorithm over them while objects in P_R may contain some objects that are not skylines. Objects in P_R are discarded (cleaned) if found to be dominated. Each dominated object in P_R represents an *unnecessary request* by our framework as its expensive attribute is requested even though it is not in the final answer. Cleaning P ensures that (a) the remaining objects in P are guaranteed to be in the final preference answer and (2) P is as small as possible, which leads to more efficient pruning operation (discussed shortly). Fifth, and finally, we prune objects from the set Q that are guaranteed not to be in the final answer regardless of their e-values. Each object $q \in Q$ is considered for two pruning cases, each progressively more expensive. Case 1: q is pruned if its bounding value v is less than or equal to U. This is an $O(1)$ operation (per object) that compares U to an object's bounding value. If this case holds, we can guarantee that q is dominated by its *dominating object* over *all* attributes. Case 2: q is compared against each object $p \in P$ and pruned if dominated by p over cheap attributes. This operation is $O(m)$ (per object), where m is the cardinality of P. The pruned object q is guaranteed never to be in the final preference answer as (a) it is dominated over cheap attributes by an object in P and (b) it is guaranteed to be dominated over its expensive attribute, as all objects in P have better e-values than all objects in Q. It is clear that the efficiency of our framework relies on the punning ability of Phase II. The more we prune the more we save by avoiding *unnecessary requests* for e-values.

Example. Figure 3(b) provides an example of a range request for $U = 10$ that returns object i. Pruning set P consists of $P_R = \{i\}$ and $P_{S_c} = \{g\}$. Here, objects c, d, and j (all part of set Q) have a bounding values equal to $U = 10$. These objects are pruned in *Case 1* as they are guaranteed to be dominated by g (their dominating object). Objects a and h are pruned in *Case 2* after comparison with P. Only object e survives both pruning cases.

Cost. The cost of Phase II is affected by how the range request deals with *overlapping objects* from S_c. Let $|R|$ be the cardinality of the objects that are returned by the range request, and $|S_c|_{<U}$ be those objects in S_c with expensive attribute values less than U. Given our assumption of a stateless request (Section 3), objects in $|S_c|_{<U}$ that were requested in Phase I would also be returned by the range request of Phase II. To handle overlapping objects, we pass the id of each object in $|S_c|_{<U}$ to the expensive source to imply the e-values are not needed for these objects. The cost of transmitting the id values is $|S_c|_{<U}(cost_t)$. The second part of the cost model accounts for the computation and transmission of e-values for non-overlapping objects. The cost of this retrieval (in addition to the overlapping object id transmission) is $(|R| - |S_c|_{<U})(cost_t + cost_c)$. For our running example (Figure 3(b)), $|S_c|_{<U} = 0$ and $|R| = 1$, thus the cost is $0 + 1(cost_t + cost_c)$.

5.1.3 Phase III: Cleaning

Phase III is responsible for computing the final preference answer by cleaning any remaining objects in Q that cannot be preferred objects. Up to this point, objects in S_c and P are guaranteed skyline answers, while objects in Q may or may not be skyline objects. Phase III involves two steps.

First, a request step makes a random request to source E for objects in Q. If possible, we also request that E only return the objects such that their e-values are less than S_{c-max}, the maximum e-value over all objects in S_c, as objects in Q with e-values *greater* than S_{c-max} are guaranteed to be dominated by any object in S_c.

Second, a computation step cleans objects in Q that are not preference answers. Similar to the cleaning step in Phase II, we seed the initial skyline result with objects in S_c and P. We then invoke any skyline algorithm over objects in Q. Any cleaned objects in Q represent *unnecessary requests* by our framework. After cleaning, all objects in S_c, P, and Q represent the final preference answer.

Example. The final answer $\{b, e, f, g, i\}$ for our running example in Figure 3 (b) would attempt to clean $Q = \{e\}$ using an the known skyline set as $\{b, f, g, i\}$.

Cost. The cost of e-value retrieval in Phase III is influenced by the random-access request for all non-pruned incomplete objects in Q. Let $|D|$ be the size of the entire data set, $|NP|_Q$ be the number of objects pruned from Q during Phase II, and $|Q|_{>S_{c-max}}$ be the number of objects in Q that have e-values greater than S_{c-max}. The cost of Phase III retrieval is $(|D| - (|S_c| + (|R| - |S_c|_{<U}) + |Q|_{>S_{c-max}} + |NP|_Q))(cost_t + cost_c)$. For our running example (Figure 3(b)), $|D| = 10$, $|Q|_{>S_{c-max}} = 0$, and $|NP|_Q = 5$, thus the cost is $(10-(3+(1-0)+0+5))(cost_t+cost_c)$, or simply $1(cost_t + cost_c)$.

5.1.4 Choosing a U Value

The pruning effectiveness of Phase II depends widely on the range request determined by the value U. Inherently, U determines the pruning set P, which in turn prunes objects from Q, thereby reducing the number of *unnecessary requests*. The theoretical best value for U creates a set P such that two *pruning objectives* are met: (1) All objects in P are part of the final answer set and (2) P is able to prune all objects from Q. If both objectives are met, no *unnecessary requests* will be made, and there is no need for Phase III, as all objects in P are in the final answer and all excessive objects in Q are pruned. However, achieving both objectives is not always possible. For space reasons, we omit proof of this claim. However, it is straightforward to construct an example where it is not possible to find a U value that achieves both objectives.

We propose five heuristic methods to derive U such that we come as close as possible to meeting both pruning objectives.

MAX. The MAX method selects U as the *maximum* e-value of the objects in S_c. For the running example data in Figure 3, the MAX method would set $U = 80$ (the *DTime* value of object f). The intuition behind this method is that the range request is guaranteed to retrieve e-values for all objects in the final preference answer. Since U is the maximum value in S_c, any object q whose value is *not* returned *cannot* be in the final preference answer, as (1) q is dominated over the subspace C by at least one object in S_c and (2) q's e-value is guaranteed to be dominated, as it is worse than all e-values in S_c. Thus, using MAX implies Phase III is not needed. The drawback of MAX is that it potentially retrieves e-values for many extraneous objects that are instantly discarded. This method also causes the greatest amount of overlap between Phase I and Phase II e-value requests, as all objects in S_c are retrieved by the range request.

MIN. The MIN method selects U as the *minimum* e-value of the objects in S_c. The example in Figure 3 uses the MIN method, as ($U = 10$). The advantage of MIN is that at least *one* returned object is guaranteed to be in the final preference answer, as its e-value will dominate all the e-values for objects in S_c. Also, the MIN method causes no overlap between Phase I and Phase II e-value requests, as $|S_c|_{<U} = 0$. A potential drawback is that a small set of objects may be returned, thus Q may not be pruned effectively.

MOSTDOM. The MOSTDOM method selects U as the e-value of the object p in S_c that is found to dominate the *most* objects during Phase I. Given that p is a "strong" dominating object, the intuition behind this method is that objects with an e-value less than p will have a greater chance of being in the final preference answer, i.e., they will not be dominated by the e-value of p. To find p, we use of the disjoint set data structures (discussed in Section 5.1.1) to quickly find the *dominating count* for each object in S_c.

BOUNDMAX. The BOUNDMAX method selects U as the *maximum* boundary value of the objects not in S_c. The goal of this method is to get a *tighter* maximum than the MAX method. For instance, if the object p in S_c used for the MAX method does is *not* found to dominate any objects, this implies all objects not in S_c are dominated by an object with an e-value v that is *less* than p's e-value. The BOUNDMAX method derives this value v.

BOUNDMIN. The BOUNDMIN method selects U as the *minimum* boundary value of the objects not in S_c. This method is similar in nature to BOUNDMAX, except it attempts to find a *minimum* value based on boundary values.

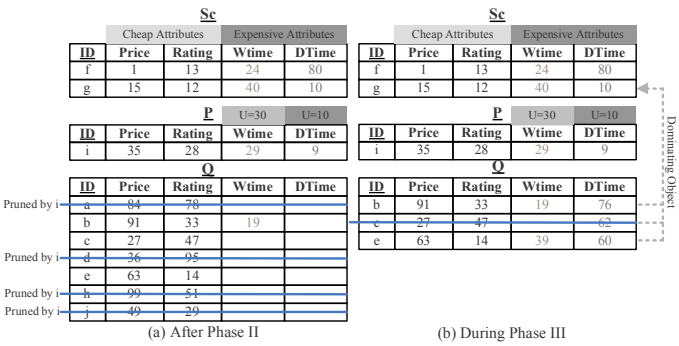


Figure 4: Skyline evaluation over multiple expensive attributes

5.2 Multiple Expensive Attributes

This section discusses the case of skyline preference evaluation over multiple expensive attributes. The framework is similar to that of the single expensive-attribute case. Thus, we outline the unique features of the multiple expensive-attribute framework. For illustrative purposes, we use the running example in Figure 4 that depicts the data in Figure 1 where attributes *Price* and *Rating* are "cheap", while attributes *WTime* and *DTime* are expensive.

5.2.1 Phase I: Initialization

Phase I is similar to that of the single expensive-attribute framework, where we compute S_c over all "cheap" attributes. The only difference is that we request the expensive attributes values (e-values) for all objects in S_c from *each* of the s expensive sources, instead of a single expensive attribute.

Example. In Figure 4 (a), S_c is comprised of objects f and g, thus we retrieve the *WTime* and *DTime* attributes for both objects, highlighted in gray.

5.2.2 Phase II: Pruning

Instead of computing a single U value in the case of the single expensive attribute case, this phase begins by computing a different upper-bound U value $[U_1, ..., U_s]$ for each expensive attribute. Each U value may be different, and calculated independent of one another. A range request is then made to each expensive source using the corresponding U value. After this request, a pruning set P is created that contains objects that are *complete*, i.e., have *all* of their attribute values (both cheap and expensive). A set Q is then formed that consists of *incomplete* objects, i.e., objects that do *not* have all of their expensive attribute values. Note that objects in Q may have had *some* of their e-values returned from the range request. Each object $q \in Q$ is then compared to each object in P and pruned in a manner similar to the single expensive-attribute framework.

Example. In Figure 4(a), with $U_1 = 30$ for attribute *WTime* and $U_2 = 10$ for attribute *DTime*, the range requests will retrieve *WTime* values 19 and 29 for objects b and i, respectively, and *DTime* value 9 for object i. After this retrieval, object i is the only object that forms set P. Finally, the pruning step discards objects a, d, h, and j from set Q.

5.2.3 Phase III: Cleaning

Similar to the single expensive attribute case, Phase III makes a random request for expensive attributes in Q, except to *each* expensive attribute source such that the e-values are *less than* or *equal* to the corresponding maximum e-value in S_c. Unlike the simpler skyline case, objects can *still* be incomplete after this random request. At this point, Phase III proceeds with a *fill-in* step, where a request is made for each incomplete attribute of objects that are known to have at *least* one e-value that is less than its boundary value. If this condition holds, the object has a *chance* of being in the preference answer based on its superiority in that attribute compared to its dominating object. Otherwise, the object is instantly discarded as: (1) if the complete expensive attribute is *greater* than it's boundary value, it is known to be dominated, and (2) the incomplete expensive attributes cannot dominate *any* object in S_c, as the attributes are guaranteed to be *greater* than the maximum attribute value in S_c. After this fill-instep, all objects in S_c, P, and Q are complete, and Phase III finished by cleaning Q similar to the single expensive attribute case, yielding a final answer.

Example. In Figure 4(b), the initial random request for attribute *DTime* returns values 76, 62, and 60 for objects b, c, and e, respectively, while value *WTime* value 39 is returned for object e. Objects b, c, and e all have g as a dominating object in S_c. In this case, c has a boundary values of 10 for attribute *DTime*, which is better than its *actual DTime* values of 62. Thus, c can be discarded immediately as (1) its *DTime* values is known to be dominated by object g and (2) its *WTime* values *cannot* dominate any other object in P or S_c, as it would have been returned during Phase II or earlier random request for Phase III. Finally, the cleaning step removes object e from Q to yield the final skyline answer $\{b, f, g, i\}$

6. MULTI-OBJECTIVE QUERY PROCESSING

This section presents our framework applied to the multi-objective preference method [1]. We first discuss the query processing case for a single expensive attribute, then discuss multiple expensive attributes.

6.1 Single Expensive Attribute Case

This section covers a multi-objective preference evaluation framework involving a single expensive attribute. We focus on the multi-objective case that combines a single cheap attribute (abbr. C_{val}) with the single expensive attribute (abbr. E_{val}) using a monotone ranking function $F(C_{val}, E_{val})$. Given our assumption that local computation is essentially "free" (Section 3), the multi-objective case that combines two cheap attributes can be solved using the skyline framework, as the monotone function can be executed cheaply on-the-fly during runtime. The input to the multi-objective framework involves a data set D, a reference to "cheap" attributes C, a reference to the expensive attribute source E, and a monotone function F that specifies the two attributes it combines (i.e., C_{val} and E_{val}).

6.1.1 Phase I: Initialization

Unlike the skyline framework, Phase I of the multi-objective framework functions over the *independent* cheap attributes. *Independent* attributes are those that do not take part in the multi-objective function. A skyline is run over these cheap attributes to form S_c. Objects in S_c are guaranteed to be in the final preference answer. A *random request* is then made to source E that retrieves the expensive attributes for all objects in S_c. After, function $F(C_{val}, E_{val})$ is applied to each object in S_c.

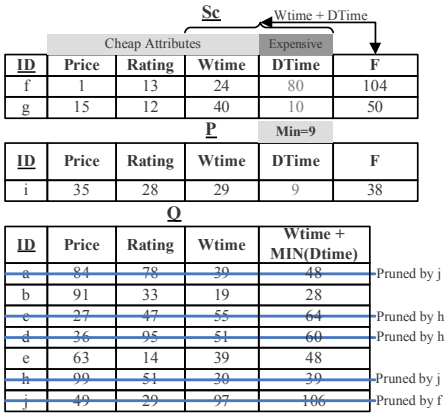


Figure 5: Multi-objective evaluation over a single expensive attribute

Example. Figure 5 gives an example of the set S_c for the data in Figure 1, where the cheap attributes *Price* and *Rating* are independent, while we apply function F (the sum) to cheap attribute *WTime* and expensive attribute *DTime*. The expensive *DTime* values are highlighted in gray, while the combined attribute F (*WTime*+*DTime*) is given in the right-most table column.

6.1.2 Phase II: Pruning

Phase II does *not* derive a U value like the skyline framework, and begins by making a request to E for the *minimum* e-value over *all* objects in D (abbr. *MIN(E)*), along with its object id. We note this request is *not* sorted access (a main assumption in Section 3), as we only request the minimum value, and no sorted values thereafter. The object with value *MIN(E)* is marked as P, and may or may not be an object already in S_c. If P is *not* an object in S_c, it is compared with each object in S_c and discarded if found to be dominated.

Like the skyline framework, Phase II then assigns all *incomplete objects* to a set Q. However, pruning rules for the multi-objective framework are different. Each object q in set Q goes through a pruning step as follows: (1) calculate the function value F for q using $F(C_{val}, MIN(E))$, this is a *lower* bound on q's function value, then (2) check if q is dominated by P or any object in S_c over all dimensions. If q is found to be dominated, it is instantly discarded. Intuitively, the lower bound value $F(C_{val}, MIN(E))$ of q represents the *best* functional value for q since it uses the minimum e-value $MIN(E)$ over all objects. Thus, if q is dominated by an object in S_c or P over all attributes, it can be safely discarded.

Example. Consider our running example in Figure 5 where *MIN(E)*=9, meaning object i is marked as P. Further, object i is not dominated by any object in S_c, thus is not discarded. After retrieving i, all objects in Q have functional values $WTime + MIN(DTime)$. Only objects b and e survive pruning, while all other objects in Q are marked with the object that caused them to be pruned.

6.1.3 Phase III: Cleaning

Like the skyline framework, Phase III begins by first making a *random request* to source E for objects remaining in Q. Unlike the skyline case, Phase III then generates a final preference answer by (1) computing the value $F(C_{val}, E_{val})$ for objects in Q and (2) cleaning objects from P and Q using S_c as the initial seed of Note that sets P *and* Q are cleaned, as objects in P may be dominated once all e-values for Q have been retrieved.

Example. We depict the final multi-objective preference answer $\{f, g, i\}$ for our running example in Figure 5, found by cleaning Q=$\{b,e\}$ and $P = \{i\}$, using $\{f,g\}$ as the initial seeded answer.

6.2 Multiple Expensive Attributes

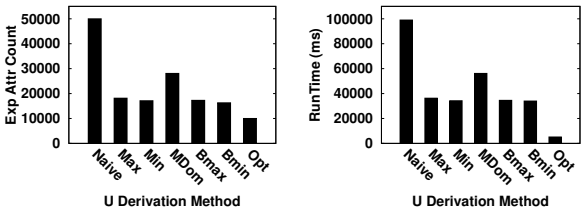
We now discuss multi-objective queries over multiple expensive attributes. Our solution covers the case where a single cheap attribute C_{val} is combined with s expensive attributes E_1,..., E_s using function $F(C_{val}, E_1, ..., E_s)$. Due to space, we do not discuss the case where s cheap attributes C_1,...,C_s are combined with a single counterpart expensive attribute $E_1, ..., E_s$ as $F(C_1, E_1)$,..., $F(C_s, E_s)$, as it can be solved using a framework similar to that from Section 6.1.

6.2.1 Phase I: Initialization

Phase I, similar to the single expensive attribute case, calculates S_c over the independent cheap attributes, except it requests *all* expensive attributes for this set.

6.2.2 Phase II: Pruning

Phase II diverges from the case of a single expensive attribute. First, the phase begins by requesting the *minimum* e-value for *each* expensive attribute (abbr. $MIN(E_1), ..., MIN(E_s)$). Each object with a minimum expensive attribute is assigned to set P. A random request is then made to *fill in* each missing value for all objects in P. At this point, the function $F(C_{val}, E_1, ..., E_s)$ is calculated for each object in P, and P is cleaned by seeding the initial preference answer to S_c. Similar to the single expensive attribute case, we create a set Q that contains all incomplete objects. However, we then apply an *iterative* pruning process to Q. Iterative pruning progressively tightens the lower-bound function value for each object $q \in Q$. Object q is pruned at any time if it is found to be dominated by objects in S_c or P. In the initial iteration, all objects $q \in Q$ are assigned a lower-bound function value F of $F(C_{val}, MIN(E_1), ..., MIN(E_s))$. Each object q is then pruned through comparison with S_c and P much like the single expensive attribute case. In the next iteration, Phase II refines the lower bound function value accuracy for each remaining object $q \in Q$ by selecting an expensive attribute E_1 to retrieve for each object q, where E_1 is the same attribute for *all* objects. Intuitively, E_1 should be the attribute with minimum retrieval cost ($cost_t + cost_c$) over all outstanding expensive attribute sources. We then update the function value for each object q as $F(C_{val}, E_1, MIN(E_2)..., MIN(E_s))$. At this point, another pruning operations is applied to all objects $q \in Q$. The iterative request-then-prune steps of Phase II end when either (1) Q is empty or (2) only a *single* expensive attribute has yet to be retrieved.



(a) Attribute retrievals (b) Runtime

Figure 6: U-Value options for skyline framework

6.2.3 Phase III: Cleaning

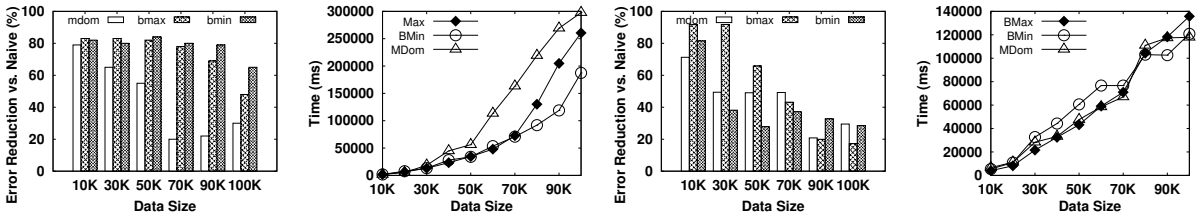
Phase III, similar to the other frameworks, computes the final preference answer by first making a random request for the remaining incomplete expensive attribute for objects in Q. At this point, objects in Q are complete. Set P and Q are then cleaned to yield a final answer.

7. EXPERIMENTAL EVALUATION

We now experimentally evaluate the performance of our frameworks for skyline and multi-objective query evaluation. Our experiments involve the following implementations. (1) Our skyline framework for single and multiple expensive attributes. Both skyline implementations have five variants based on the U-value derivation method (presented in Section 5.1.4). Each variant is identified by the following subscript: min, max, mdom (for MOSTDOM), bmax (for BOUNDMAX), and bmin (for BMIN). (2) Our multi-objective framework for single and multiple expensive attributes. Note that unlike skylines, the multi-objective framework do not have variants. (3) Since there is no related work that applies to our problem directly, we implement two alternative frameworks for skyline and multi-objective query processing for comparison purposes: (a) Naive. This framework mirrors the naive implementation outlined in Section 4. (b) Opt. This is an optimal framework that is of theoretical interest only, as it always make the optimal number of requests for expensive attributes based on a priori knowledge of their values. All frameworks use the BNL skyline algorithm [4] for processing cheap attributes and pruning, as this is the most straightforward method of executing skylines in an real DBMS.

All frameworks are implemented in the back-end executor (query processor) of the PostgreSQL 8.3.5 open-source database [21]. The experiment machine is an Intel Core2 8400 at 3Ghz with 4GB of RAM running Ubuntu Linux 8.04. Cheap attributes are stored locally in relational tables, while expensive attributes exist on a *remote* machine (details shortly). Thus, extensive changes were made to the Postgres query processor in order to request then materialize third-party remote data. Our two main performance metrics are *remote requests*, counting the number of expensive attributes returned during query processing, and *elapsed time* reported by the PostgreSQL EXPLAIN ANALYZE command.

We make use of two data sets in our experiments. (1) Synthetic data using the generator specified in [4]. Unless otherwise mentioned, this data contain six attributes, generated independently of one another. Dataset sizes range from 10K to 100K tuples, with the default set as 50K. To model remote attributes for this dataset, we built a custom web service running on a separate machine connected through a stan-



(a) SEA Error Reduction (b) SEA Runtime (c) MEA Error Reduction (d) MEA Runtime

Figure 7: Scalability of skyline framework

dard 10Base-T Ethernet local area network. (2) *Real* web data for restaurants in the Minneapolis, MN area, containing five attributes. Three attributes were downloaded and stored locally: (a) *ZPrice* and (b) *Service* rating, both taken from the Zagat website [28]. (c) *CPrice*, taken from the Citypages website [9]. Two remote attributes were accessed in real-time through actual web service: (d) *Rating*, accessed through the Yelp review web service [27], and (e) *Driving Time*, accessed through the Micosoft MapPoint web service [19].

7.1 Skyline Framework

This section studies the effectiveness of our skyline frameworks for single and multiple expensive attribute cases as presented in Sections 5. We first evaluate the U-value derivation methods for the skyline framework. We then evaluate the scalability of both frameworks.

7.1.1 U-Value

Figure 6 gives the results for different U-value derivation methods over the default 50K data set for the single expensive attribute framework. Figure 6(a) depicts the number of expensive attribute retrievals needed for the five methods presented in Section 5.1.4, along with the counts for the optimal and naive skyline algorithms. The naive method makes a total of 50K requests, as *each* object requires an e-value retrieval. Meanwhile, the optimal algorithm requires 10K retrievals, mirroring the number of skyline objects. Many of the U-value methods bring the skyline framework close to the optimal number of requests. In this case, min requires only 17,110 requests and represents the best performing method. This number represents an 82% improvement over the naive method, i.e., min makes 82% *less* unnecessary requests than the naive method. This percentage is measured as $\frac{Count_{naive}-Count_{min}}{Count_{naive}-Count_{opt}}$, where the denominator represents the amount of unnecessary requests made by the naive method, and the numerator represents the difference in the request count between the naive method and min method. Meanwhile, mdom requires 28083 requests, representing the worst case of all the methods, but still a 54% improvement over the naive method. Figure 6(b) gives the runtime for each of the five U-value methods, along with the runtime of the optimal and naive implementations. The runtime of each U-value is clearly correlated with its number expensive attribute retrievals. The optimal implementation exhibits superior performance. The performance discrepancy between the optimal and other U-value methods is due to our implementation approach. The optimal method runs a skyline in a *single* pass, and then performs the optimal number of requests. Thus, it does not execute the three phases of our proposed framework.

7.1.2 Scalability

Figure 7(a) gives the expensive attribute request error reduction percentage for the bmin, bmax, mdom methods in the single expensive attribute (abbr. SEA) framework as the data size increases from 10K to 100K. For readability, we omit max and min, as they performed similarly to bmax and bmin. Clearly, bmin and bmax scale well, and consistently reduce unnecessary requests. This performance is due to the pruning effectiveness of Phase II. The error reduction drops quickly for mdom as the data size grows. This drop occurs because as data sizes grow, it is more likely an object in Q will be dominated by *more* than one object in LA. Thus, it is less likely the object chosen by mdom will have a U-value capable of pruning effectively. Figure 7(b) provides the runtimes for each method. We omit Sky_{naive}, as it did not scale well. In fact, for larger data sets, its runtime was close to an hour. The runtime for each method mirrors its pruning effectiveness given in Figure 7(a).

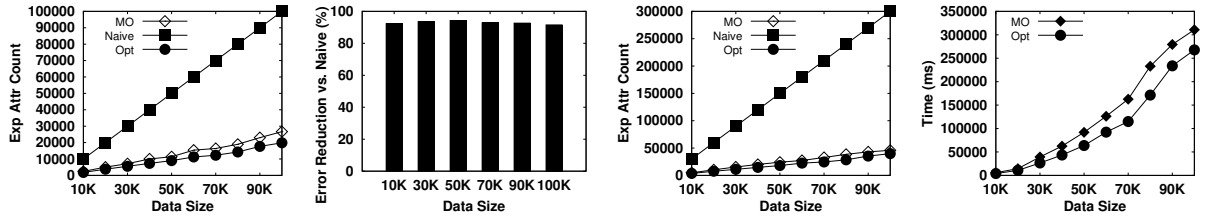
Figure 7(c) gives the expensive attribute request error reduction percentage for the multiple expensive attribute framework (abbr. MEA, again omitting min and max). Unless otherwise noted, the default number of expensive attributes is three. Also, the U-value derivation method is the same across all expensive attributes. It is clear that the error reduction percentage of each variant drops as the data size grows. However, even for larger data sizes the percentage remains at 20%. A main reason for this performance lies in a property of the data set: since each attribute value is generated independent of one another, none of the U-values are able to adequately retrieve a near-optimal pruning set as the data size grows. Figure 7(d) provides the run-times for each method. Each method shows similar linear performance. We also note that the run-times are *less* than the single expensive attribute framework. The reason for this performance is that *more* objects are pruned in the multi-expensive attribute case, meaning less requests in Phase III.

7.2 Multi-Objective Framework

This section evaluates the effectiveness our multi-objective frameworks for the case of single and multiple expensive attributes presented in Sections 6.

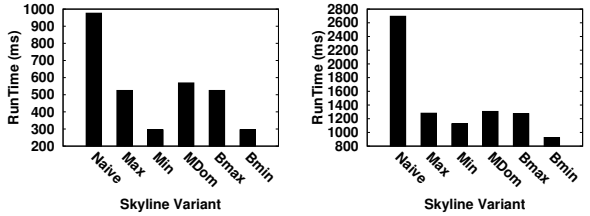
7.2.1 Scalability

Figure 8 gives the results for the multi-objective frameworks as the input data size ranges from 10K to 100K. Figure 8(a) plots the number of expensive attributes retrieved by the single expensive attribute (abbr. SEA) framework compared to the optimal and naive methods. The pruning effectiveness of our approach is clear in this case, as our framework shows an almost identical retrieval count as the optimal method. The pruning effectiveness is also depicted in Figure 8(b), that plots the percentage improve-



(a) SEA Retrievals (b) SEA Error Reduction (c) MEA Retrievals (d) MEA Runtime

Figure 8: Scalability of multi-objective framework



(a) Retrieval count (b) Runtime

Figure 9: Skyline framework over real data

ment over the naive method (percentage calculation covered in Section 7.1.1). Meanwhile, Figures 8(c) and 8(d) plot the expensive attribute retrieval count and runtime for the multiple expensive attribute (abbr. MEA framework. In both cases, we see performance in line with the optimal method.

7.3 Real Web Data

Figure 9 gives the expensive attribute retrieval count and runtime for our skyline framework variants when run over our real data set consisting of restaurants in the greater Minneapolis area. Each variant of our framework is far superior to the naive implementation. The main reason for our superior performance is the correlation inherent in the data. For instance, a high *CPrice* from Citypages review source will likely imply a high *ZPrice* from the Zagat review. In this case, pruning becomes more effective because objects dominated in the "cheap" attribute subspace will likely not become final skylines, as their expensive attributes are inferior to those of the objects in S_c. Indeed, for our real experimental data, the "cheap" *Service* rating attribute and expensive overall *Rating* attribute are correlated, causing more objects to be pruned. We can conclude from this experiment that our framework performs very well with real-world, correlated data.

8. CONCLUSION

This paper presented a framework for evaluating both skyline and multi-objective preference queries involving expensive attributes. We first defined expensive attributes as any attribute requiring non-trivial computation/retrieval cost during query runtime. We then presented our framework capable of processing skyline and multi-objective queries in the case of both single and multiple expensive attributes. The efficiency of the framework is due to the ability to *discard* objects *without* retrieving their expensive attributes, thus reducing high computational overhead in query processing. Our proposed framework is novel, in the sense that we only assume random and range access to expensive attributes: a model that translates well to real data domains. Our experiments, implemented in an actual database and run over synthetic and live web data, prove that our frameworks are efficient, and provide superior performance to current database implementations that must evaluate *every* expensive attribute to process preference queries.

9. REFERENCES

[1] W.-T. Balke and U. Güntzer. Multi-objective Query Processing for Database Systems. In *VLDB*, 2004.

[2] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT*, 2004.

[3] I. Bartolini, P. Ciaccia, and M. Patella. Salsa: computing the skyline without scanning the whole sky. In *CIKM*, 2006.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, 2001.

[5] N. Bruno, L. Gravano, and A. Marian. Evaluating Top-k Queries over Web-Accessible Databases. In *ICDE*, 2002.

[6] K. C.-C. Chang and S. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.

[7] L. Chen and C. Lian. Dynamic skyline queries in metric spaces. In *EDBT*, 2008.

[8] J. Chomicki et al. Skyline with presorting. In *ICDE*, 2003.

[9] Citypages Restaurants: http://www.citypages.com/restaurants/.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 21, pages 498Ű–524. MIT Press and McGraw-Hill, second edition, 2001.

[11] K. Deng, X. Zhou, and H. T. Shen. Multi-source Skyline Query Processing in Road Networks. In *ICDE*, 2007.

[12] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, 1993.

[13] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in MANETs. In *ICDE*, 2006.

[14] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[15] A. Kemper et al. Optimizing disjunctive queries with expensive predicates. In *SIGMOD*, 1994.

[16] W. Kießling and G. Köstler. Preference SQL: Design, Implementation, Experiences. In *VLDB*, 2002.

[17] G. Koutrika and Y. Ioannidis. Personalization of Queries in Database Systems. In *ICDE*, 2004.

[18] E. Lo, K. Y. Yip, K.-I. Lin, and D. W. Cheung. Progressive skylining over Web-accessible databases. *Data and Knowledge Engineering*, 57(2), 2006.

[19] Microsoft MapPoint: http://www.microsoft.com/virtualearth/.

[20] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, 2003.

[21] PostgreSQL: http://www.postgresql.org.

[22] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *VLDB*, 2006.

[23] K. Stefanidis, E. Pitoura, and P. Vassiliadis. Adding Context to Preferences. In *ICDE*, 2007.

[24] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2), 2000.

[25] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD*, 1998.

[26] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu. Efficient skyline query processing on peer-to-peer networks. In *ICDE*, 2007.

[27] Yelp: http://www.yelp.com.

[28] Zagat: http://www.zagat.com/.