Supporting Exploratory Queries in Databases

Abhijit Kadlag[1], Amol V. Wanjari[1], Juliana Freire[2], and Jayant R. Haritsa[1]

[1] Dept. of Computer Science & Automation
Indian Institute of Science, Bangalore 560012, INDIA
{abhijit,amol,haritsa}@csa.iisc.ernet.in
[2] Computer Science & Engineering
OGI/OHSU, Beaverton, Oregon 97006, USA
juliana@cse.ogi.edu

Abstract. Users of database applications, especially in the e-commerce domain, often resort to exploratory "trial-and-error" queries since the underlying data space is huge and unfamiliar, and there are several alternatives for search attributes in this space. For example, scouting for cheap airfares typically involves posing multiple queries, varying flight times, dates, and airport locations. Exploratory queries are problematic from the perspective of both the user and the server. For the database server, it results in a drastic reduction in effective throughput since much of the processing is duplicated in each successive query. For the client, it results in a marked increase in response times, especially when accessing the service through wireless channels.
In this paper, we investigate the design of automated techniques to minimize the need for repetitive exploratory queries. Specifically, we present SAUNA, a server-side query relaxation algorithm that, given the user's initial range query and a desired cardinality for the answer set, produces a relaxed query that is expected to contain the required number of answers. The algorithm incorporates a range-query-specific distance metric that is weighted to produce relaxed queries of a desired shape (e.g., aspect ratio preserving), and utilizes multi-dimensional histograms for query size estimation. A detailed performance evaluation of SAUNA over a variety of multi-dimensional data sets indicates that its relaxed queries can significantly reduce the costs associated with exploratory query processing.

1 Introduction

Users of database applications, especially in the e-commerce domain, often resort to exploratory "trial-and-error" queries since the underlying data space is huge and unfamiliar, and there are several alternatives for search attributes in this space [1]. Consider, for example, the query interface provided at Travelocity [2], a popular Web site for travel planning. Here, for each itinerary, users must select origin and destination airports, departure and return times, departure and return dates, and may optionally select airlines. Faced with this environment, users often pose a *sequence* of *range queries* while planning their travel schedule. For example, the first query could be:

```
SELECT * FROM FLIGHTS WHERE
DepartureTime BETWEEN 10.00 A.M. AND 11.00 A.M. AND
DepartureDate BETWEEN 09-11-2003 AND 09-12-2003 AND
Origin = "LAX" AND Destination = "JFK" AND Class = "ECONOMY".
```

© Springer-Verlag Berlin Heidelberg 2004

and if the result for this query proves to be unsatisfactory, it is likely to be followed by

```
SELECT * FROM FLIGHTS WHERE
DepartureTime BETWEEN 08.00 A.M. AND 12.00 A.M. AND
DepartureDate BETWEEN 09-11-2003 AND 09-13-2003 AND
Origin = "LAX" AND Destination = "JFK" AND Class = "ECONOMY".
```

and so on, until a satisfactory result set is obtained.

Such trial-and-error queries are undesirable from the perspective of both the user and the database server. For the server, it results in a drastic reduction in effective throughput since much of the processing is duplicated in each successive query. For the client, it results in a marked increase in response times, as well as frustration from having to submit the query repeatedly. The problem is compounded for users who access the Web service through a handheld device (PDA, smart-phone, etc.) due to the high access latencies, cumbersome input mechanisms, and limited power supply.

Too Few Answers

A primary reason for the user dissatisfaction that results in repetitive queries is the *cardinality* of the answer set – the Web service may return *no* or insufficiently few answers, and worse, give no indication of how to alter the query to provide the desired number of answers [1]. (The complementary problem of "too many answers" has been previously addressed in the literature – see, for example [3].)

Two approaches, both implemented on the *client-side*, have been proposed for the "too few answers" problem: The 64K Inc.[4] engine augments query results (if any) with statistical information about the underlying data distribution. Users are expected to utilize this information to rephrase their queries appropriately. However, it is unrealistic to expect that naïve Web users will be able (or willing) to perform the calculations necessary to rephrase their queries.

An alternative approach was proposed in Eureka [1]. In response to the initial user query, Eureka caches the relevant portion of the *database* at the client machine, allowing follow-up exploratory queries to be answered locally. A major drawback is that the user needs to install a customized software for each of the Web services that she wishes to access. In addition, this strategy may not be feasible for resource-constrained client devices which may be unable to host the entire database segment, or which are connected through a low-bandwidth network.

Finally, yet another possibility is to convert the user's range query into a point query (e.g., by replacing the box represented by the query with its centerpoint) and then to use one of the several Top-K algorithms available in the literature (e.g., [5]) with respect to this point. However, this approach is unacceptable since it runs the risk of *not providing all the results that are part of the original user query*. Further, as discussed later in this paper, *closeness to a point may not be equivalent to closeness to the query box*.

The SAUNA Technique

In this paper, we propose **SAUNA** (Stretch A User query to get N Answers), a *server-side* solution for efficiently supporting exploratory queries. More formally, given an initial user query Q^I (which we expect to return M answers), and given the desired number of answers N, if $N > M$, SAUNA derives a new relaxed query Q^R which *contains* Q^I and is expected to have N answers. A pictorial representation of a SAUNA relaxation is shown in Figure 1 for a two-dimensional range query.

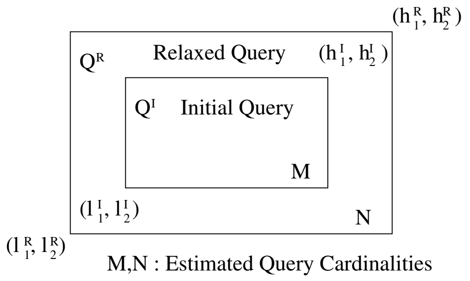


Fig. 1. Range query relaxation in 2 dimensions

Note that a variety of relaxed queries, which may even be infinite in number, could be derived that obey the above constraints. In this solution space, SAUNA aims to deliver a relaxed query that (a) minimizes the distance of the additional answers with respect to the original query, that is, it aims to derive the closest $N - M$ answers, and (b) minimizes the data processing required to produce this set of answers. The first goal is predicated on defining a distance metric for points lying outside the original query – this issue is well understood for *point-queries* [5] but not for the *range* (or *box*) queries that we consider here. Therefore, SAUNA incorporates a box-query-specific distance metric that is suitably weighted to produce relaxed queries of a desired shape (e.g., aspect-ratio preserving with respect to the original query). To achieve the second goal, SAUNA utilizes multi-dimensional histograms as the tool for query size estimation. Histograms [6,7] are the de facto standard technique for maintaining statistical summaries in current database systems, and therefore our system is easily portable to these platforms. While uni-dimensional histograms are currently the norm, techniques for easily building and maintaining their multi-dimensional counterparts have recently appeared in the literature [8].

In an overall sense, SAUNA extends the work of [5] which was limited to *point* queries, to the more general and complex class of *range* queries. As we show in Section 5, a detailed performance evaluation of SAUNA over a variety of real and synthetic multi-dimensional data sets stored on a Microsoft SQL Server 2000 engine indicates that SAUNA's relaxed queries can significantly reduce the costs associated with exploratory query processing, and in fact, often compare favorably with the *optimal-sized* relaxed query (obtained through off-line processing). Further, these improvements are obtained even when the memory budget for storing statistical information is extremely limited.

2 Problem Definition

We model the data space as being characterized by D dimensions with the corresponding attribute set being $\{X_1, X_2, \ldots, X_D\}$. For ease of exposition, we assume that all attribute domains are normalized to the range [0,1].

The initial query posed by the user is a D-dimensional hyper-rectangle defined by $Q^I = \{[l_1^I, h_1^I], [l_2^I, h_2^I], \ldots, [l_D^I, h_D^I]\}$ where each l_i^I and h_i^I denote the lower and upper limit of the query along the ith dimension (see Figure 1). That is, $0 \leq l_i^I \leq h_i^I \leq 1$, $\forall i\ 1 \leq i \leq D$. Here, some attributes will have *ranges* (i.e., $l_i^I < h_i^I$), some will be *points* (i.e., $l_i^I = h_i^I$), and some will be *don't-cares* (i.e., $l_i^I = 0, h_i^I = 1$). We assume that the user specifies the attributes that are *fixed* in that they should not be relaxed.

The relaxed query is denoted by $Q^R = \{[l_1^R, h_1^R], [l_2^R, h_2^R], \ldots, [l_D^R, h_D^R]\}$, with $Q^I \subseteq Q^R$ and $0 \leq l_i^R \leq l_i^I$ and $1 \geq h_i^R \geq h_i^I$, $\forall i\ 1 \leq i \leq D$. The differences $r_{il} = l_i^I - l_i^R$ and $r_{ih} = h_i^R - h_i^I$ $(r_{il}, r_{ih} \geq 0)$ are used to denote the relaxations w.r.t. the lower and upper limits of the original query along the ith dimension.

Along with the query, the user also provides N, the desired cardinality of the answer set. The estimated cardinalities of the original and relaxed queries are denoted by $M = \mid Q^I \mid$ and $N' = \mid Q^R \mid$, respectively. Relaxation is invoked only if $M < N$, and the goal of the relaxation system is to produce a relaxed query such that (a) $N' \geq N$, (b) $N' - N$ is minimized, (c) the additional $N - M$ answers returned to the user are the *closest neighbors* of Q^I, and (d) the data processing required to produce these additional answers is minimized. The definition of closest neighbors is made precise in the next section.

3 Distance Metrics for Box Queries

Most distance functions used in practice are based on the general theory of *vector p-norms* [9], with $1 \leq p \leq \infty$. For example, $p = 2$ is the classical Euclidean metric, $p = 1$ represents the Manhattan metric, and $p = \infty$ results in the Max metric. In the remainder of this paper, for ease of exposition, we assume that all distances are measured with the Euclidean metric. Note, however, that the SAUNA relaxation algorithm can be easily adapted to any of the alternative metrics.

3.1 Reference Points

When computing the distances of database tuples with respect to *point* queries, it is clear that the distances are always to be measured (whatever be the metric) between the pair of points represented by the database tuple and the point query. However, when we come to *box* (range) queries, which is the focus of this paper, the issue is not so clear-cut since it is not obvious as to which point in the box should be treated as the reference point. In fact, it is even possible to think of distances being measured with respect to *a set* of reference points.

One obvious solution is to take some point inside the box (e.g., the center), treat the box as being represented by this point, and then resort to the traditional distance measurement techniques. However, this formulation appears highly unsatisfactory since the spatial structure of the box, which is representative of the user intentions, is completely ignored. Instead, we contend here that the user's specification of a box query implies that she would prefer answers that are *close to the periphery* of the box. To motivate this, consider the example situation shown in Figure 2, where point P is farther from the box center, O, than point Q i.e., $r_2 > r_4$, but P's distance from the closest face of the box is smaller than the corresponding distance for Q i.e., $r_1 < r_3$. In this situation, we expect the user to prefer point P over Q since there is less deviation with respect to the complete box. The above observation can be formally captured by the following reference point

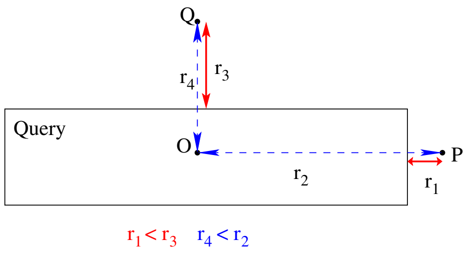


Fig. 2. Measuring distance from periphery. P is closer to periphery than Q

assignment technique: For measuring the distance between a point P and a query box B, the reference point on B is the point of intersection of the perpendicular line drawn from P to the nearest face or corner of the box B.

In summary, given a point $P = \{p_1, p_2, ..., p_D\}$ and a box-query B with lower and upper limits $l_i(B)$ and $h_i(B)$ respectively, we denote the component of distance on the i-th dimension as

$$\begin{aligned} d_i(P,B) &= p_i - h_i(B) \quad if \;\; p_i > h_i(B) \\ &= l_i(B) - p_i \quad if \;\; p_i < l_i(B) \\ &= 0 \qquad\qquad\quad otherwise \end{aligned}$$

and the overall (Euclidean) distance between P and B as

$$dist(P,B) = \sqrt{\sum_{i=1}^{D} (d_i(P,B))^2} \tag{1}$$

Note that with this formulation, all points that lie *within* or *on* the box have an associated distance of zero.

3.2 Attribute Weighting

An implicit assumption in the above discussion is that relaxation on all dimensions is equivalent. However, it is quite likely that the user finds relaxation on some attributes more desirable than on others. For example, a business traveler may be time-conscious as compared to price, whereas a vacationer may have the opposite disposition. Therefore, we need to *weight* the distance on each dimension appropriately. That is, we modify Equation 1 to

$$dist(P,B) = \sqrt{\sum_{i=1}^{D}(d_i(P,B) * w_i)^2} \qquad (2)$$

where $w_i \geq 0$ is the weight assigned to dimension i.

One option certainly is to explicitly acquire these weights from the user, and use them in the above equation. However, as a default in the absence of these inputs, we resort to the following: *Use the box shape as an indicator of the user's intentions*. Specifically, we can assume that the user is willing to accept a relaxation on each range dimension that is *proportional* to the range size in that dimension, i.e., the user would prefer what we term as an *Aspect-Ratio-Preserving* relaxation (this metric preserves the aspect ratio of the user-supplied query, hence the name). This objective can be easily implemented by setting the weights

$$w_i^{aspect} = \frac{1}{Asp\_ratio(i)} = \frac{Max_{i=1}^{D}(h_i(B) - l_i(B))}{h_i(B) - l_i(B)}$$

An alternative interpretation of the user's box-query structure could be that attributes should be relaxed in *inverse* proportion to their range sizes, since the user has already *built in* relaxation into the larger ranges of her query. This can be implemented with the following weighting function

$$w_i^{inverse} = Asp\_ratio(i) = \frac{h_i(B) - l_i(B)}{Max_{i=1}^{D}(h_i(B) - l_i(B))}$$

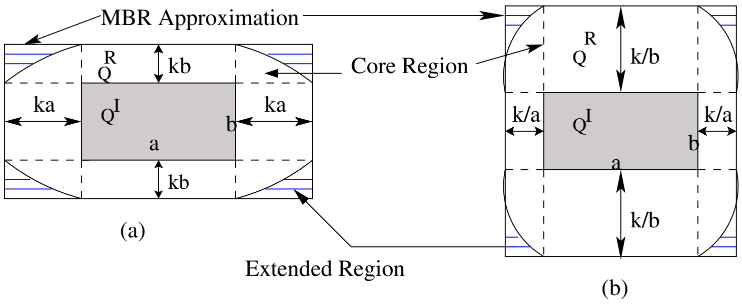


Fig. 3. Distance Metrics and Relaxation regions: (a) Aspect (b) Inverse

Figure 3 shows an example of the relaxed queries produced by using the *Aspect* and *Inverse* metrics, respectively. Given a constant k and relaxation units a and b (in the x and y axes, respectively), we see in these figures that the locus of points equidistant from the original query is not hyper-rectangular in the corners. Therefore, we approximate the relaxed queries by their *Minimum Bounding (Hyper)-Rectangles* since only hyper-rectangular queries are supported in core SQL, discounting special features such as UDF (User Defined Functions). The area enclosed within the locus is referred to as the *core* region and the area between the *core* region and the MBR rectangle is called the *extended region*.

As a final point, note that if the user has specified a *point* query as opposed to a box query, then the above formulation degenerates to a traditional Top-N query [5], where the goal is to find the nearest N neighbors to the query point.

4 The SAUNA Relaxation Algorithm

SAUNA, our new query relaxation technique, attempts to ensure the desired cardinality and quality of answers for user queries while simultaneously trying to reduce the cost of relaxed query execution. Specifically, the relaxation algorithm generalizes to box queries the approach taken for point queries in [5].

Histograms are used for query size estimation in the SAUNA relaxation process. In particular, we use *multi-dimensional histograms* for the experiments reported in this study. Although such histograms have been previously touted as being resource-intensive to create and maintain, recent work [8] has addressed this problem by proposing an online adaptive mechanism for easily building and maintaining multi-dimensional histograms, the so-called self-tuning histograms.

Due to their summary nature, histograms can provide only estimates, and not the exact values. Therefore, when relaxing a query to produce N answers, there is always a risk of either under-estimating or over-estimating the cardinality of the answer set. While under-estimation results in inefficiency due to accessing more database tuples than necessary, over-estimation requires the query to be relaxed further and submitted again – a *restart* in the terminology of [5].

Estimation strategies possible in this environment include a conservative approach that completely eliminates restarts at the risk of getting many more tuples than necessary, and an optimistic approach that trades restarts for improved efficiency. These *No-Restarts* and *Restarts* approaches were implemented in [5] by assuming that all database tuples in a histogram bucket are at the maximum or minimum distance, respectively, with respect to the point query. In Figure 4, we present the MinDist and MaxDist algorithms to find these minimum and maximum distances, respectively for range queries. Both these algorithms are linear in the number of query attribute dimensions. We describe below the various relaxation strategies for box queries that are based on these distance computations. The range query counterparts of the *NoRestart* and *Restart* approaches in [5] are termed here as *Box-NoRestart* and *Box-Restart*, respectively. For more details, we refer the reader to [10].

Box-Dynamic Strategy. Since Box-Restarts and Box-NoRestarts represent extreme solutions, an obvious question is whether an intermediate solution that provides the best of both worlds can be devised? For this, we adopt the dynamic workload-based mapping strategy of [5], which attempts to find the relaxation distance that minimizes

```
$Algorithm\ MinDist\ (Box\ q, Bucket\ b,$
        $Metric\ metric)$ {
$Point\ Nearest,\ Nearest^l,\ Nearest^h;$
$\forall\, i : 1 \leq i \leq D$
$begin$
$Nearest_i^l = q_i^l \quad if\ b_i^l \leq q_i^l \leq b_i^h$
    $= b_i^l \quad if\ q_i^l < b_i^l$
    $= b_i^h \quad otherwise$
$Nearest_i^h = q_i^h \quad if\ b_i^l \leq q_i^h \leq b_i^h$
    $= b_i^l \quad if\ q_i^h < b_i^l$
    $= b_i^h \quad otherwise$
$if\ |q_i^l - Nearest_i^l| < |q_i^h - Nearest_i^h|$
    $Nearest_i = Nearest_i^l$
$else$
    $Nearest_i = Nearest_i^h$
$end\ \forall\, i$
$return\ dist_{metric}\ (Nearest,\ q)$
}
```

(a) MinDist

```
$Algorithm\ MaxDist\ (Box\ q, Bucket\ b,$
        $Metric\ metric)$ {
$Point\ Farthest,\ Farthest^l,\ Farthest^h;$
$\forall\, i : 1 \leq i \leq D$
$begin$
$Farthest_i^l = b_i^l \quad if\ q_i^l \leq b_i^l$
    $= b_i^h \quad otherwise$

$Farthest_i^h = b_i^l \quad if\ q_i^h \leq b_i^l$
    $= b_i^h \quad otherwise$

$if\ |q_i^l - Farthest_i^l| > |q_i^h - Farthest_i^h|$
    $Farthest_i = Farthest_i^l$
$else$
    $Farthest_i = Farthest_i^h$
$end\ \forall\, i$
$return\ dist_{metric}\ (Farthest,\ q)$
}
```

(b) MaxDist

Fig. 4. Algorithms for computing distances

the expected number of tuples retrieved for a set of queries while ensuring a reduced number of restarts. This is implemented as follows: Given α as a parameter such that

$$d_q(\alpha) = d_q^{BR} + \alpha\left(d_q^{BNR} - d_q^{BR}\right)$$

where d_q^{BR} and d_q^{BNR} are the *Box-Restarts* and *Box-NoRestarts* distances for query q, we need to find the value of $d_q(\alpha)$ that minimizes the average number of tuples retrieved for a given query workload. Since $d_q(\alpha)$ is a unidimensional function of α, the *golden search* algorithm [11] can be utilized to estimate this optimal value of α. Note that this approach requires an initial "training workload" to determine a suitable value of α, which can then be used in the subsequent "production workloads".

Relaxation Algorithm. While the Box-Dynamic strategy does reduce the likelihood of restarts, it does not completely eliminate them. To ensure that we do not get into a situation where there are repeated restarts of a given query, we follow the strategy that if the Box-Dynamic strategy happens to fail for a particular query, then we immediately resort to the conservative Box-NoRestarts strategy – that is, all queries are relaxed with at most one restart. The complete sequence of steps of the SAUNA relaxation algorithm is shown in Figure 5. The input parameter W to the algorithm is the set of weights to be used in case we wish to have dimension-specific relaxations.

```
Algorithm SAUNA Relaxation (Query $Q^I$, Integer $N$, Weights $W$)
{
1  $M$ = estimateCardinality($Q^I$);
2  if $M < N$
3      $Q^R$ = relaxBoxDynamic($Q^I$, $N$, $W$);
4      numAnswers = execute($Q^R$);
5      if numAnswers $\geq N$ return the $N$ nearest answers;
6      else
7          $Q^{R'}$ = relaxNoRestart($Q^I$, $N$, $W$);
8          execute($Q^{R'}$);
9      endif
10 else
11     numAnswers = execute($Q^I$);
12     if numAnswers $\geq N$ return all answers;
13     else
14         $M$ = numAnswers;
15         go to Step 7;
16     endif
17 endif
18 return
}
```

Fig. 5. SAUNA relaxation algorithm

5 Performance Evaluation

We have conducted a detailed performance evaluation of the SAUNA technique and a representative set of results are presented here – for full details see [10].

5.1 Experimental Settings

In our experiments, we used a variety of synthetic and real-world data sets – these datasets are the same as those used in [5]. The real-world data sets consisted of the US census data set ($199,523$ tuples) and the Forest data set ($581,012$ tuples) obtained from [12]. We selected from these data sets the same set of attributes as [5]. The synthetic data consisted of the *Gauss* and *Array* data sets, each containing $500,000$ tuples [5].

The experiments were performed using multidimensional equidepth histograms [6], as they are both accurate and simple to implement. Further, an N-dimensional unclustered concatenated-key B^+-tree multidimensional index covering all the query attributes was built over each data set.

The query workload consists of queries with the number of range dimensions varying from 2 to 4, which is typical of many e-commerce applications. The specific queries were generated by moving a query template over the entire domain space, returning a set of 100 queries. All results we report are averages for this set of hundred queries.

To serve as comparative yardsticks for SAUNA's performance, we used two benchmarks:

Sequential (SEQ) : In this strategy, a sequential scan of the database is made in order to produce a sorted list of the tuples w.r.t. their distance from the query box, after which the top N tuples are returned.

Optimal (OPT) : This strategy refers to a hypothetical optimal relaxation strategy which produces the *minimally relaxed query* that contains the desired answer set. Note that even the minimum bounding hyper-rectangle enclosing the N nearest tuples of a query box is not guaranteed to return N answers only, and often returns more than N answers. In our experiments, the answers for OPT were found through an offline complete scan of all the data tuples.

For the results shown here, the experimental settings were number of desired answers $N = 10$, *Aspect* distance metric, and number of histogram buckets $= 256$; the *gauss* and *array* datasets were generated with a zipfian parameter $z = 1$.

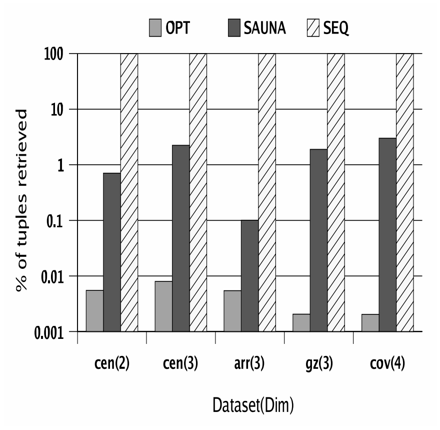
Our experiments were conducted on a Pentium IV machine running the Windows 2000 operating system.

5.2 SAUNA Performance

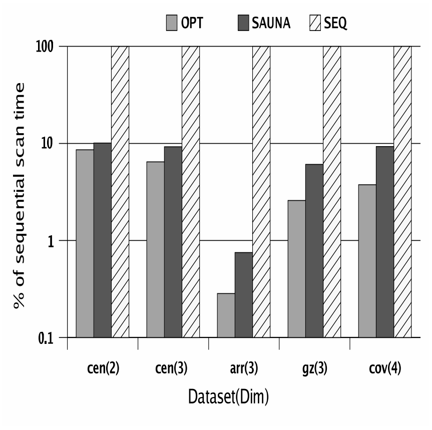
The performance of SAUNA as compared to SEQ and OPT is shown in Figure 6(a) with respect to the number of tuples retrieved (the Y-axis is shown on a *log scale*), for the various datasets. The labels in the X-axis: cen, cov, arr and gz refer to the census dataset, cover dataset, array dataset and gauss dataset, respectively, while the number in brackets refers to the number of data dimensions. Note that the data is clustered on the disk and hence the percentage of tuples retrieved is indicative of the number of disk accesses.

The first point to observe here is that for all the datasets, SAUNA requires processing less than 4% of the tuples – in fact, for the *census* and *array* datasets they are less than 1%. Secondly, note that there is a substantial difference between the optimal performance and that of SAUNA. This is due to the fact that SAUNA relies on statistical information that is limited by a tight memory budget (only 256 histogram buckets, consuming around 5KB memory, were used in this experiment). The performance gap between SAUNA and OPT is considerably larger for the *gz* and *cover* datasets as compared to the *array* and *census* datasets (this behavior was also seen in our other experiments). We attribute this to the dense and clustered nature of the *gz* and *cover* datasets which results in retrieval of a large number of tuples even from a small query space. Again this is largely dependent on the quality of histograms available.

In Figure 6(b), we show the running times of the SAUNA and OPT strategies (excluding the time required to find the optimal relaxed query), normalized to the execution time of SEQ, for the various datasets (the Y-axis is shown on a *log scale*). The first point to note here is that the SAUNA execution times are below 10% of the sequential scan time for all the datasets. Secondly, for the *census* and *array* datasets the SAUNA times are close to that of OPT, and even for the other datasets the difference is not much. The number of query restarts were found to be negligible for *census* and *array* datasets. About 10% of the queries required restarts for the *gz* and *cover* datasets, which we attribute to the skew in these datasets.



(a) Percentage of tuples retrieved



(b) Execution time

Fig. 6. SAUNA Performance

The execution time figures clearly indicate the efficiency of SAUNA w.r.t. the optimal strategy. Again, it should be noted that it is not the relaxation algorithm, but the quality of the histograms (the type and number of buckets) that affect the efficiency of SAUNA as compared to the optimal in terms of number of tuples retrieved or the execution time.

Besides different datasets, we also evaluated the performance of SAUNA with respect to (a) varying N, the desired result cardinality; (b) varying the skew in the data; (c) varying the distance metric; and, (d) varying the number of buckets in the histogram. Due to space constraints, we refer the reader to [10] for more details. An interesting feature that we observed is that the performance of SAUNA w.r.t. OPT, improved dramatically with increased values of N. Thus in environments where a higher number of answers are expected (e.g., in a banking application where the manager wants to see a list of 250 customers with balance more than $\$100,000$), we expect SAUNA to perform even better. Further, the performance of SAUNA is robust in that even with heavily-skewed data, it retrieved less than 3% of the tuples to achieve the desired relaxation. Also we found that the performance characteristics for *Inverse* distance metric are very similar to those of the *Aspect* metric.

Overall, our experiments show that SAUNA, despite being constrained by the limited memory resources, robustly and efficiently provides automated query relaxation. When more memory is provided, the performance improves accordingly.

6 Conclusions

In this paper, we proposed SAUNA, a novel server-based framework for automated query relaxation that improves the efficiency and efficacy of query exploration over large and unknown data spaces. Unlike previous approaches that are limited to point queries, SAUNA is able to relax multi-dimensional range queries. Through the use of an intuitive range-query-specific distance metric, SAUNA returns high-quality answers that are *closest* to the user-specified query box. In addition, the SAUNA framework can be easily integrated with commercial RDBMS that support histograms.

Our experimental results indicate that SAUNA significantly reduces the costs associated with exploratory query processing, and in fact, often compares favorably with the optimal-sized relaxed query Further, these improvements are obtained even when the memory budget for storing statistical information is extremely limited. Specifically, even with as low a memory budget as 5 KB, SAUNA was able to provide satisfactory relaxation retrieving less than 10% of the tuples in the database and taking less than 10% of the time taken by sequential scan.

Acknowledgements. This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science & Technology, Govt. of India.

References

1. J. Shafer and R. Agrawal. Continuous querying in database-centric web applications. *Computer Networks*, 33(1-6):519–531, 2000.
2. Travelocity. `http://www.travelocity.com`.
3. M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of SIGMOD Conf.*, pages 219–230, 1997.
4. 64K Inc. DBGuide introduction and technology overview, 1997.
5. N. Bruno, S. Chaudhuri and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS*, 27(2), 2002.
6. M. Muralikrishna and D. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. of SIGMOD Conf.*, pages 28–36, 1998.
7. V. Poosala, Y. Ioannidis, P. Haas and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. of SIGMOD*, pages 294–305, 1996.
8. A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proc. of SIGMOD Conf.*, pages 181–192, 1999.
9. I. Gradshteyn and I. Ryzhik. *Tables of Integrals, Series and Products*. Academic Press, 2000.
10. A. Kadlag, A. Wanjari, J. Freire and J. Haritsa. Supporting Exploratory Queries in Databases. Technical Report, TR-2003-02, DSL/SERC, 2003.
    `http://dsl.serc.iisc.ernet.in/pub/TR/TR-2003-02.pdf`.
11. W. Press et al. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, 1993.
12. UCI knowledge discovery in databases archive.
    `http://kdd.ics.uci.edu/summary.data.type.html`.