Efficient Quantile Retrieval on Multi-Dimensional Data*

Man Lung Yiu[1], Nikos Mamoulis[1], and Yufei Tao[2]

[1] Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong, {mlyiu2,nikos}@cs.hku.hk

[2] Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong, taoyf@cs.cityu.edu.hk

Abstract. Given a set of N multi-dimensional points, we study the computation of ϕ-quantiles according to a ranking function F, which is provided by the user at runtime. Specifically, F computes a score based on the coordinates of each point; our objective is to report the object whose score is the ϕN-th smallest in the dataset. ϕ-quantiles provide a succinct summary about the F-distribution of the underlying data, which is useful for online decision support, data mining, selectivity estimation, query optimization, etc. Assuming that the dataset is indexed by a spatial access method, we propose several algorithms for retrieving a quantile efficiently. Analytical and experimental results demonstrate that a branch-and-bound method is highly effective in practice, outperforming alternative approaches by a significant factor.

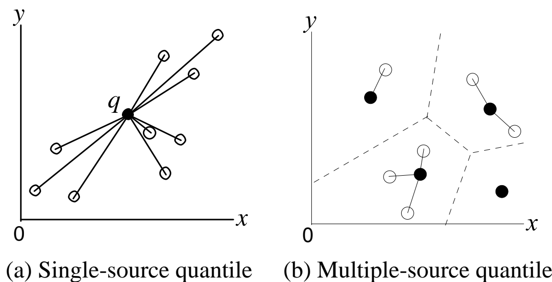
1 Introduction

We study quantile computation of a *derived measure* over multi-dimensional data. Specifically, given (i) a set P of N points in d-dimensional space, (ii) a continuous function $F : \mathbb{R}^d \rightarrow \mathbb{R}$, and (iii) a value $\phi \in [0, 1]$, a quantile query retrieves the ϕN-th smallest F-value of the objects in P. For instance, the median corresponds to the 0.5-quantile, whereas the maximum is the 1-quantile. Quantiles provide a succinct summary of a data distribution, finding application in online decision support, data mining, selectivity estimation, query optimization, etc.

Consider a mobile phone company that has conducted a survey on customers' preferences regarding their favorite service plans. The two dimensions in Figure 1a capture two properties of a monthly plan (e.g., the price and amount of air-time); each white point represents the preferences of a customer on these properties. Assume that the company is planning to launch a new plan corresponding to the black dot q. To evaluate the potential market popularity of q, the manager would be interested in the distribution of the similarity between q and customers' preferences. For this purpose, F may be defined as the Euclidean distance between q and a white point and quantiles for various values of ϕ could be retrieved. As another (spatial) example, assume that point q in Figure 1a is a pizza shop and the white points correspond to residential buildings. The *median* residential building distance (from the pizza shop) might be useful to the shop owner, in order to plan adequate number of staff for pizza delivery.

* Work supported by grants HKU 7380/02E and CityU 1163/04E from Hong Kong RGC, and SRG grant (Project NO: 7001843) from City University of Hong Kong.

The query in Figure 1a is a *single-source* query because the ordering of data points depends on one source only. A more complex scenario is shown in Figure 1b, where the white points correspond to residential areas, and the black dots represent supermarkets. Each dashed polygon is the "influence region" [21] of a supermarket, which covers those residential areas that find it as the nearest supermarket. A market analyst would like to obtain the distribution of the distance from a residential area to its nearest supermarket, in order to decide a suitable location to open a new supermarket. A quantile query in this case is a *multiple-source* one, because the ordering of the white points is determined by multiple sources (the supermarkets).



(a) Single-source quantile (b) Multiple-source quantile

Fig. 1. Examples of quantiles based on derived Euclidean distances

Our goal is to compute a quantile by accessing only a fraction of the dataset, *without knowing the ranking function F in advance*. Previous research [1, 14, 15, 9, 8] in the database community focused on computing/maintaining approximate quantiles, whereas we aim at obtaining the *exact* results. Furthermore, our problem is completely different than the so-called "spatial quantiles" in computational geometry [12, 5]. For example, a *spatial center* is a location p (not necessarily an actual point in the dataset) such that every hyperplane containing p defines a "balanced" partitioning of the dataset (i.e., the numbers of points in the two partitions differ by less than a threshold). Our quantile, on the other hand, is a *one-dimensional* value in the output domain of F.

We develop several solutions that leverage a multi-dimensional index (e.g., R-tree) to prune the search space, starting with a variation of the incremental nearest neighbor (INN) algorithm [11]. This algorithm is not efficient as its cost linearly increases with ϕ. We then present a faster algorithm, which iteratively approximates the F-distribution using linear functions. Our last solution combines the branch-and-bound framework with novel heuristics that minimize the I/O cost based on several problem characteristics. We analyze the relative performance of the proposed approaches, theoretically and experimentally. Finally, we generalize our algorithms to other variations of the problem including progressive retrieval, batch processing, etc.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the problem, while Section 4 presents the details of the proposed algorithms, and analyzes their performance. Section 5 extends our solutions to other types of quantile retrieval. Section 6 contains an experimental evaluation, and finally, Section 7 concludes the paper with directions for future work.

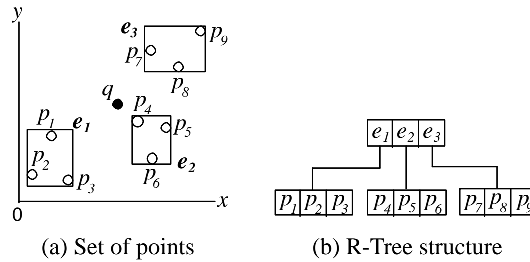
2 Related Work

We treat quantile computation of a derived measure as a spatial query. We review indexing and query evaluation for multidimensional data in Section 2.1. Section 2.2 surveys existing methods for retrieving quantiles on non-indexed data.

2.1 Spatial query processing

R-trees [10] have been extensively used for indexing spatial data. Figure 2a shows a set of points on the 2D plane, indexed by the R-tree in Figure 2b. The R-tree is balanced and each node occupies one disk page. Each entry stores the MBR (minimum bounding rectangle) that encloses all spatial objects in the corresponding sub-tree. Leaf nodes store object MBRs and their record-ids in the spatial relation that stores the objects. R-trees were originally designed for spatial range queries, but they can be used to process more advanced spatial queries, like nearest neighbors [11], spatial joins [4], reverse nearest neighbors [21], skyline queries [18], etc.

The aggregate R-tree (aR-tree) [13, 17] is a variant of the R-tree where each entry is augmented with an aggregate measure of all data objects in the sub-tree pointed by it. The aR-tree was originally designed for the efficient evaluation of spatial aggregate queries, where measures (e.g., sales, traffic, etc.) in a spatial region (e.g., a country, a city center) are aggregated. In Section 4, we show how aR-trees, augmented with `COUNT` measures, can be exploited for efficient quantile computation.



(a) Set of points (b) R-Tree structure

Fig. 2. R-Tree example

Our work is also relevant to nearest neighbor (NN) queries. A NN query asks for the point closest to an input point q. INN, the state-of-the-art algorithm for NN search [11], retrieves the nearest neighbors of q *incrementally* in ascending order of their distance to q. INN browses the R-tree and organizes the entries of the visited nodes in a min-heap based on their distances to q. First, all root entries are enheaped. When a non-leaf entry is dequeued, its child node is accessed and all entries in the child node are enheaped. When a leaf entry (data object) is dequeued, it is guaranteed to be the next nearest neighbor.

2.2 Quantile computation

The computation of a quantile is known as the selection problem in the theory community. [19] is an overview of theoretical results on the selection problem. Sorting the elements is a straightforward solution but this requires $O(N \log N)$ comparisons. An early algorithm in [3] needs only $O(N)$ comparisons but it may access some elements

multiple times. It was shown in [16] that $O(N^{\frac{1}{t}})$ memory is necessary and sufficient for solving the selection problem in t passes of data.

This theoretical result implies that the *exact* quantile (selection) problem can only be answered in multiple passes with limited memory. Hence, the database community attempted to compute *approximate* quantiles with only one pass of the data. An element is an ϵ-approximate ϕ-quantile if its rank is within the range $[(\phi-\epsilon)N, (\phi+\epsilon)N]$. [1, 14, 15, 9, 8] presented algorithms for retrieving an approximate quantile with limited memory in at most one pass of data. The best result [9] requires only $O(\frac{1}{\epsilon}\log(\epsilon N))$ memory. Recently, [6] studied computation of biased (extreme) quantiles in data streams, which requires less space than the upper bound given by [9]. Observe that the memory required by the above methods is at least proportional to $1/\epsilon$. Thus, they are not appropriate for computing exact quantiles (or approximate quantiles with very small ϵ).

In our problem setting, the ranking function F is dynamic and only known at runtime. Due to this dynamic nature, pre-materialized results may not be used. In addition, we aim at utilizing existing indexes to minimize the data required to be accessed in order to compute the quantile, whereas existing techniques [1, 14, 15, 9, 8, 6] operate on non-indexed, one-dimensional data. Their focus is the minimization of error, given a memory budget, where one pass over the data is essential. The problem of computing quantiles on indexed one-dimensional data (i.e., by a B$^+$-tree) is not interesting, since the high levels of the tree are already an equi-depth histogram that can be easily used to derive the quantiles efficiently. On the other hand, there is no total ordering of multi-dimensional data, thus R-trees cannot be used directly for ad-hoc spatial ranking.

A viable alternative for handling dynamic ranking functions in multidimensional data is to maintain a random sample from the dataset and compute an approximate quantile value from it. It is known [2] that, for any random sample of size $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$, the ϕ-quantile of the sample is also an ϵ-approximate quantile of the dataset with probability at least $1-\delta$. The number of required samples directly translates to the required memory size. Thus, random sampling technique is inadequate for retrieving exact quantiles (where $\epsilon = 0$) or approximate quantiles with very small ϵ. In this paper, we propose *index-based* methods for efficient computation of *exact* quantiles on derived ranking measures over multidimensional data.

3 Problem Definition

Let P be a set of N d-dimensional points and $F : \mathbb{R}^d \rightarrow \mathbb{R}$ be a continuous function on the domain of P. The continuity property implies that points close together have similar F values. The coordinate of a point p along the i-th dimension is denoted by $p(i)$. Given a real value ϕ in the range $[0, 1]$, a ϕ-quantile query returns the ϕN-th smallest F value among all points in the dataset. Without loss of generality, we assume that ϕN is an integer.

We assume that the dataset is indexed by a `COUNT` aR-tree. Each entry in the tree is augmented with the number of points in its subtree. The ranking function F is application-dependent. Moreover, we require two bounding functions F_l and F_u, which take the MBR of a non-leaf entry e as input and return the range $[F_l(e), F_u(e)]$ of possible F values for any point in it. Given an entry e of the aR-tree, the derived range $[F_l(e), F_u(e)]$ is used by our algorithms to determine whether the entry can be

pruned or not from search. Computation of tight F_l and F_u bounds is essential for good query performance. Although our discussion assumes aR-trees, our framework is also applicable to other hierarchical spatial indexes (where non-leaf nodes are augmented with aggregate information [13]).

Our aim is to provide a generic framework for processing quantile queries using aR-trees. In the following, we provide examples of four ranking functions F. The first two define *single-source* quantile queries and take one (or zero) parameter (e.g., a query point). The last two define *multiple-source* quantile queries with multiple parameters.

Distance ranking Each object in a dataset is associated with a rank based on its distance from a reference query point q. For an MBR e, we have $F_l(e) = mindist(q, e)$ and $F_u(e) = maxdist(q, e)$; the minimum and maximum distances [11], respectively, between any point in e and q.

Linear ranking A linear function combines coordinate values of a point into a single score. Such a function is the generalization of the SUM function used in top-k queries [7]. Given d weights $w_1, w_2, \cdots, w_d$, the ranking function F is defined as $F(p) = \sum_{i\in[1,d]} w_i \cdot p(i)$. For an MBR e, we have $F_l(e) = \sum_{i\in[1,d]} w_i \cdot e^{\vdash}(i)$ and $F_u(e) = \sum_{i\in[1,d]} w_i \cdot e^{\dashv}(i)$, where $e^{\vdash}(i)$ and $e^{\dashv}(i)$ are the lower and upper bounds of the extent of e on the i-th dimension.

Nearest-site distance ranking This scenario is a generalization of simple distance ranking. We consider the ranking of objects based on their distances from their nearest query point in a given set. Given query points (sites) $q_1, q_2, \cdots, q_m$, the ranking function F is defined as $F(p) = \min_{i\in[1,m]} dist(q_i, p)$, where $dist$ denotes the distance function (e.g., Euclidean distance). We have $F_l(e) = \min_{i\in[1,m]} mindist(q_i, e)$ and $F_u(e) = \min_{i\in[1,m]} maxdist(q_i, e)$, for an MBR e. We assume that the number m of query points is small enough to fit in memory. For example, the data points represent users and the query points represent facilities (e.g., restaurants in the town).

Furthest-site distance ranking Unlike the previous example, we consider the ranking of objects based on their distances from their furthest query point in a given set. For instance, a small group of people (modeled as query points) decide to meet at the same restaurant. The maximum distance of the restaurant from the group reflects their meeting time. Given query points (sites) $q_1, q_2, \cdots, q_m$, the ranking function F is defined as $F(p) = \max_{i\in[1,m]} dist(q_i, p)$, where $dist$ denotes the distance function (e.g. Euclidean distance). For an MBR e, we have $F_l(e) = \max_{i\in[1,m]} mindist(q_i, e)$ and $F_u(e) = \max_{i\in[1,m]} maxdist(q_i, e)$. As in the previous example, we assume that the number m of query points is small enough to fit in memory.

4 Quantile Computation Algorithms

In this section, we propose three quantile computation algorithms that apply on an aR-tree. The first method is a simple extension of a nearest neighbor search technique. The second solution is based on iterative approximation. Section 4.3 discusses an optimized branch-and-bound method for computing quantiles. Finally, a qualitative cost analysis of the algorithms is presented in Section 4.4.

4.1 Incremental search

The *incremental* quantile computation algorithm (INC) is a generalization of the incremental nearest neighbor (INN) algorithm [11]. It simply retrieves the point with the next lowest F value until the ϕN-th object is found. A pseudocode for INC is shown in Figure 3. The algorithm employs a min-heap H for organizing the entries e to be visited in ascending order of their $F_l(e)$ value. A counter cnt (initially 0) keeps track of the number of points seen so far. First, all entries of the root node are enheaped. When an entry e' is deheaped, we check whether it is a non-leaf entry. If so, its child node is accessed and all entries in the node are enheaped. Otherwise (e' is a leaf entry), e' is guaranteed to have the next lowest F value, and the counter cnt is incremented by 1. The algorithm terminates when the counter reaches ϕN.

```
Algorithm INC(R-tree R, Function F, Value φ)
1    cnt:=0;
2    H:=∅;
3    for each entry e ∈ R.root
4        Enheap(H, ⟨e, F_l(e)⟩);
5    while (H ≠ ∅)
6        e′:=Deheap(H);
7        if (e′ is a non-leaf entry) then
8            read the node n pointed by e′;
9            for each entry e ∈ n
10               Enheap(H, ⟨e, F_l(e)⟩);
11       else
12           cnt:=cnt + 1;
13           if (cnt = φN) then
14               return F(e′);
```

Fig. 3. The incremental search algorithm (INC)

Observe that the cost of INC is sensitive to ϕ. For large values of ϕ, many objects are accessed before the algorithm terminates. Next, we will present other quantile algorithms, which use the aggregate values stored at the high levels of the aR-tree and their performance is less sensitive to the value ϕ.

4.2 Iterative approximation

The motivation behind our second algorithm is to search the quantiles that correspond to some F values and progressively refine an approximation for the desired ϕ-quantile. Figure 4a illustrates an example distribution of ϕ as a function of F values. Clearly, we do not know every value on this graph a priori as the ranking function F is only known at runtime. Suppose we want to find the 0.25-quantile (i.e. $\phi = 0.25$). We initially obtain lower ($F_l(R)$) and upper bounds ($F_u(R)$) for the potential values of F without any cost, according to the MBR of the aR-tree R. In the figure, $F_l(R)$ corresponds to point a and $F_u(R)$ to point b. We reduce the computation of the 0.25-quantile to a numerical problem and apply *the interpolation method* [20] for solving it. The main idea is to approximate the distribution function as a straight line and *iteratively* "probe" for exact quantile values corresponding to the expected values based on the approximation, until the error is small enough for the F value to correspond to an exact quantile.

Continuing with the example, we first connect points a and b by a straight line, and take the F-coordinate λ_1 of the intersection (see Figure 4a) between the line and a horizontal line at $\phi = 0.25$ (i.e., λ_1 is our estimate of the 0.25-quantile). Then, we compute the point c on the *actual* ϕ-F curve whose F-coordinate is λ_1 (computation of c will be explained shortly). Since the ϕ-coordinate of c is smaller than 0.25 (i.e., λ_1 *underestimates* the 0.25-quantile), we (i) obtain the F-coordinate λ_2 of the intersection between the horizontal line $\phi = 0.25$ and the line connecting c, b, and (ii) retrieve the point d (Figure 4b) on the actual curve with λ_2 as the F-coordinate. As λ_2 *overestimates* the 0.25-quantile (the ϕ-coordinate of d is greater than 0.25), we perform another iteration by connecting c and d, which leads to e in Figure 4c. Since the ϕ-coordinate of e equals 0.25, the algorithm terminates.

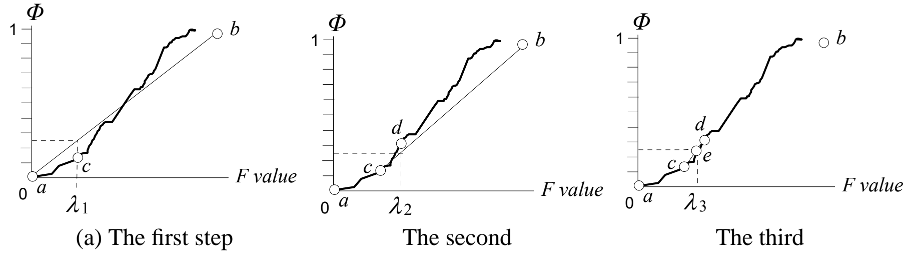


Fig. 4. Iterative approximation example

We now discuss how to use the aR-tree in order to find a point that corresponds to a value λ for F. This is done using the RANGE_COUNT function shown in Figure 5, which counts the number of data points with F values not greater than λ, while traversing the tree. The function is first invoked at the root node of an aR-tree. For an intermediate entry e, if $F_u(e) \leq \lambda$, then all the points under the subtree fall in the range and the counter is incremented by $\texttt{COUNT}(e)$, the number of points under the subtree of e. note that $\texttt{COUNT}(e)$ can directly be retrieved from the tree node containing e. Otherwise ($F_u(e) > \lambda$), if e is a non-leaf entry and $F_l(e) \leq \lambda$, then it is possible that some points under e lie in the query range. Thus, the function is called recursively to compute the remaining count of data points within the range.

Algorithm **RANGE_COUNT**(aR-tree node n, Function F, Value λ)
1 cnt:=0;
2 **for each** entry $e \in n$
3 **if** ($F_u(e) \leq \lambda$) **then**
4 cnt:=cnt+$\texttt{COUNT}(e)$;
5 **else if** ($F_l(e) \leq \lambda \wedge e$ is a non-leaf entry) **then**
6 read the node n' pointed by e;
7 cnt:=cnt+RANGE_COUNT(n', F, λ);
8 **return** cnt;

Fig. 5. Counting points in a generalized range

Figure 6 shows the pseudocode of the iterative approximation algorithm (APX). First, the bounds λ_l and λ_u of the F-range that includes the ϕ-quantile are initialized

to $F_l(R)$ and $F_u(R)$, respectively. In addition, values cnt_l and cnt_u (conservative approximations of quantiles at the end-points of the range) are initialized to 0 and N respectively. At Line 4, we link points (λ_l, cnt_l) and (λ_u, cnt_u) on the ϕ-F curve (like the one of Figure 4) by a straight line, and use the line to estimate a value λ for the ϕ-quantile. Function RANGE_COUNT is then used to compute the number cnt of points whose F value is at most λ. After that, Lines 6–9 update the coordinates for the next iteration. The loop iterates until the count cnt converges to ϕN.

Algorithm **Iterative_Approximation**(aR-tree R, Function F, Value ϕ)
1 (λ_l, cnt_l):=$(F_l(R), 0)$; // conservative lower end
2 (λ_u, cnt_u):=$(F_u(R), N)$; // conservative upper end
3 **do**
4 λ:=$\lambda_l + \frac{\lambda_u - \lambda_l}{cnt_u - cnt_l} \cdot (\phi N - cnt_l)$; // linear approximation of quantile value
5 cnt:=RANGE_COUNT($R.root$, F, λ); // actual rank for the estimated value
6 **if** $(cnt > \phi N)$
7 (λ_u, cnt_u):=(λ, cnt);
8 **else**
9 (λ_l, cnt_l):=(λ, cnt);
10 **while** $(cnt \neq \phi N)$;
11 **return** λ;

Fig. 6. The iterative approximation algorithm (APX)

4.3 Branch-and-bound quantile retrieval

Although the iterative approximation algorithm is less sensitive to ϕ, it is not very efficient as it needs to access the aR-tree multiple times. We now present a branch-and-bound algorithm for computing quantiles. Before we describe the algorithm, we introduce some notations and pruning rules employed by it.

Definition 1. *Let S be a set of aR-tree entries. For any $e \in S$, $\omega_l(e, S)$ denotes the maximum possible number of objects whose F value is at most $F_u(e)$ and $\omega_u(e, S)$ denotes the maximum possible number of objects whose F value is at least $F_l(e)$. Formally:*

$$\omega_l(e, S) = \sum_{e' \in S, F_l(e') \leq F_u(e)} \texttt{COUNT}(e') \tag{1}$$

$$\omega_u(e, S) = \sum_{e' \in S, F_u(e') \geq F_l(e)} \texttt{COUNT}(e') \tag{2}$$

Measures $\omega_l(e, S)$ and $\omega_u(e, S)$ form the basis of pruning rules for aR-tree entries during branch-and-bound traversal for quantile computation. When the context is clear, we sometimes drop the symbol S and simply use $\omega_l(e)$ and $\omega_u(e)$. To illustrate the use of these measures, consider Figure 7, showing the F range intervals of four aR-tree entries $S = \{e_1, \ldots, e_4\}$. Let $\phi = 0.5$ and $N = 50$. Note that $\omega_l(e_2) = 10 + 10 = 20 < \phi N$, which means that all objects p in the subtree of e_2 have ranks lower than the quantile. Thus, we can safely prune e_2 and avoid accessing its subtree during quantile computation. On the other hand, we cannot prune entry e_1, since $\omega_l(e_1) = 10 + 10 + 15 = 35 \geq \phi N$. Symmetrically, by computing whether $\omega_u(e_i) < (1 - \phi)N + 1$, we can determine if an entry can be pruned due to the lower ranking bound of objects in it.

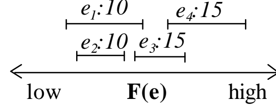


Fig. 7. Pruning example

The algorithm We can now describe in detail our branch-and-bound (BAB) quantile computation algorithm (shown in Figure 8). We initialize two variables cur_l and cur_u, which capture the number of objects guaranteed to be before and after the ϕ-quantile, respectively. The root of the aR-tree is visited and all entries there are inserted into set S. Lines 6–8 attempt to prune entries, all objects under which have rankings lower than ϕN. For this, we examine the entries ordered by their upper ranking bound $F_u(e)$. Lemma 1 (trivially proven) states the pruning condition. Lemma 2 suggests that ω_l values of entries can be incrementally computed. We can prune all entries e satisfying $F_u(e) \leq F_u(e_l^*)$ where e_l^* is the entry with the greatest F_u value satisfying the pruning condition. Lines 12–14 perform symmetric pruning; entries are examined in descending order of their lower bounds in order to eliminate those for which all indexed objects have rankings higher than ϕN. Finally, in Lines 18–21, a heuristic is used to choose a (non-leaf) entry e_c from S, visit the corresponding node, and update S with its entries. Details on how to prioritize the entries will be discussed shortly. The algorithm terminates when the requested quantile is found (Line 10 or 16).

Lemma 1. Pruning condition. *Among the objects that were pruned, let cur_l (cur_u) be the number of objects with F values smaller (greater) than the quantile object. An aR-tree entry e can be pruned if $cur_l + \omega_l(e) < \phi N$ or $cur_u + \omega_u(e) < N(1-\phi)+1$.*

Lemma 2. Efficient computation. *Let $l_1, l_2, \cdots, l_{|S|}$ ($u_1, u_2, \cdots, u_{|S|}$) be aR-tree entries in a set S such that $F_u(l_i) \leq F_u(l_{i+1})$ ($F_l(u_i) \geq F_l(u_{i+1})$). We have*
$\omega_l(l_{i+1}, S) = \omega_l(l_i, S) + \sum_{e \in S, F_u(l_i) < F_l(e) \leq F_u(l_{i+1})} \mathtt{COUNT}(e)$ *and*
$\omega_u(u_{i+1}, S) = \omega_u(u_i, S) + \sum_{e \in S, F_l(u_i) > F_u(e) \geq F_l(u_{i+1})} \mathtt{COUNT}(e)$.

Management of S We propose to use two main-memory B$^+$-trees (T_l and T_u) for indexing the entries in S; one based on their F_l values and another based on their F_u values. Insertion (deletion) of an entry takes $O(log|S|)$ time in both trees. The rationale of using the above data structure is that it supports efficient pruning of entries. Lemma 2 suggests that ω_l (ω_u) values of entries can be incrementally computed. Now, we discuss how to prune entries with rankings guaranteed to be lower than ϕN. First, we get the entry with lowest F_u value from T_u and then compute its ω_l value (by accessing entries in T_l in ascending order). Next, we get the entry with the next lowest F_u value from T_u and compute its ω_l value (by accessing entries in T_l in ascending order, starting from the last accessed location). The above process repeats until the current entry in T_u does not satisfy the pruning condition in Lemma 1. Notice that entries (in T_l and T_u) are accessed sequentially through sibling links in B$^+$-tree leaf nodes. Let e_l^* be the entry with the greatest F_u value satisfying the pruning condition. Then, we remove all leaf entries e in T_u satisfying $F_u(e) \leq F_u(e_l^*)$ and delete their corresponding entries in T_l. A symmetric procedure is applied to prune entries with rankings guaranteed to be greater than ϕN.

Algorithm **BaB_Quantile**(aR-tree R, Function F, Value ϕ)

```
1     cur_l:=0; cur_u:=0;
2     S:=∅;
3     for each entry e ∈ R.root
4         S:=S ∪ {e};
5     while (true)
6         e_l^* := entry in S with the greatest F_u value satisfying cur_l + ω_l(e_l^*) < φN ;
7         for each entry e ∈ S satisfying F_u(e) ≤ F_u(e_l^*) // pruning entries on the lower side
8             S:=S − {e}; cur_l:=cur_l+COUNT(e);
9         e_l := entry in S with the minimum F_u value;
10        if (cur_l = φN − 1 ∧ ω_l(e_l) = 1 ∧ e_l is a leaf entry) then
11            return F(e_l);
12        e_u^* := entry in S with the least F_l value satisfying cur_u + ω_u(e_u^*) < N(1 − φ) + 1 ;
13        for each entry e ∈ S satisfying F_l(e) ≥ F_l(e_u^*) // pruning entries on the upper side
14            S:=S − {e}; cur_u:=cur_u+COUNT(e);
15        e_u := entry in S with the maximum F_l value;
16        if (cur_u = N(1 − φ) ∧ ω_u(e_u) = 1 ∧ e_u is a leaf entry) then
17            return F(e_u);
18        if (φ ≤ 0.5) // heuristic for picking the next non-leaf entry to expand
19            set e_c as the non-leaf entry, overlapping e_l's F-interval, with the maximum count in S;
20        else
21            set e_c as the non-leaf entry, overlapping e_u's F-interval, with the maximum count in S;
22        S:=S − {e_c};
23        access node n' pointed by e_c;
24        for each entry e ∈ n'
25            S:=S ∪ {e};
```

Fig. 8. The branch-and-bound quantile computation algorithm (BAB)

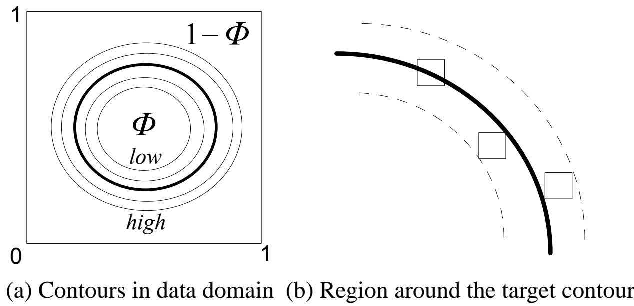
Order of visited nodes We now elaborate on which non-leaf entry should be selected for further expansion (Lines 18–21 in Figure 8); the order of visited aR-tree nodes affects the cost of the algorithm. Before reaching Line 18, entries e_l and e_u have the minimum value of $\omega_l(e_l)$ and $\omega_u(e_u)$ respectively. Intuitively, we should attempt reducing the value $\omega_l(e_l)$ (or $\omega_u(e_u)$) so that entry e_l (or e_u) can be pruned. For entry e_l, value $\omega_l(e_l)$ is determined by the count of other entries whose F-interval intersects that of the entry e_l. Thus, it suffices to identify the non-leaf entry that contributes the most to $\omega_l(e_l)$ (*maximum non-leaf component*). Lemma 3 guarantees that such a non-leaf component always exists. Similarly, we can also compute the maximum non-leaf component of $\omega_u(e_u)$. The question now is whether the removal of the maximum non-leaf component of $\omega_l(e_l)$ or that of $\omega_u(e_u)$ leads to lower overall I/O cost. Note that the algorithm terminates when the quantile result is found from either the lower side or the upper side. Hence, it is not necessary to expand non-leaf entries from both lower and upper sides. Based on this observation, a good heuristic is to select the maximum non-leaf component of $\omega_l(e_l)$ when $\phi \leq 0.5$, and that of $\omega_u(e_u)$, otherwise, in order to reach the requested quantile as fast as possible.

Lemma 3. Non-leaf component. *Let e_l (e_u) be the entry in S with the smallest ω_l (ω_u) value. There exists a non-leaf entry among all entries $e' \in S$ satisfying $F_l(e') \leq F_u(e_l)$. Also, there exists a non-leaf entry among all entries $e' \in S$ satisfying $F_u(e') \geq F_l(e_u)$.*

Proof. We will prove the first statement; the proof for the second is symmetric. Consider two cases for the entry e_l. If e_l is a non-leaf entry, then the statement is trivially true. If e_l is a leaf entry (i.e. a data point), then we have $\omega_l(e_l) > 1$ as it is not pruned before. As there are no other leaf entries with F value smaller than e_l, there must exist a non-leaf entry in S whose interval of F values intersects that of e_l.

4.4 Qualitative Cost Analysis

This section analyzes qualitatively the performance of the proposed algorithms. Our analysis considers only the number of (distinct) leaf nodes accessed by the algorithms, as the total cost is dominated by leaf node accesses (and a large enough memory buffer will absorb the effect of accessing nodes multiple times by the RANGE_COUNT queries of APX).



(a) Contours in data domain (b) Region around the target contour

Fig. 9. Analysis example

As discussed in Section 3, we assume that F is a continuous function. Figure 9a shows a set of five exemplary *contours* on the data domain. Each of them connects the set of locations having the same F value in the data domain. The continuity property of the ranking function implies that contours close together have similar F values. In our example, inner contours have lower F values than outer ones. Let F^* be the F value of the quantile corresponding to the *target contour* in bold. Note that the union of all contours with F values at most F^* enclose ϕN data points. Figure 9b magnifies the region around the target contour. Observe that any algorithm that computes the quantile correctly must access the area enclosed by dotted curves which contain all leaf nodes (rectangles) intersecting the target contour.

We now examine the *search region* of the proposed algorithms. The branch-and-bound quantile algorithm (BAB) accesses the nodes intersecting the target contour, pruning at the same time a significant fraction of nodes which do not intersect the target contour. On the other hand, the incremental search algorithm (INC) accesses all nodes intersecting contours with F values smaller than or equal to F^*. Clearly, INC is much more expensive than BAB. The iterative approximation algorithm (APX) employs an efficient range count algorithm so it only visits the nodes intersecting the target contour and a few other contours (due to multiple trials). Thus, APX is not as effective as BAB as APX accesses more space (i.e., more contours). Summarizing, BAB is expected to outperform both INC and APX in terms of I/O. Another observation is that different quantiles require different cost. This is obvious for INC. For APX and BAB, it is more

expensive to compute the median than extreme quantiles because more nodes intersect the target contour of the median than that of extreme quantiles.

5 Variants of Quantile Queries and Problem Settings

In this section, we discuss variants of quantile query evaluation and cases where aR-trees may not be available. Section 5.1 discusses how the proposed algorithms can be adapted for approximate quantile queries. Section 5.2 examines efficient computation of *batch* quantile queries. Finally, Section 5.3 investigates quantile computation for cases where the data are indexed by simple (i.e., not aggregate) R-trees or when only spatial histograms are available.

5.1 Approximate and progressive quantile computation

An ϵ-approximate quantile query returns an element from the dataset with a rank in the interval $[(\phi - \epsilon)N, (\phi + \epsilon)N]$, where N is the data size, ϕ and ϵ are real values in the range $[0, 1]$. The goal is to retrieve an accurate enough estimation, at low cost. For INC, the termination condition at Line 13 of the algorithm in Figure 3 is modified to $cnt = (\phi - \epsilon)N$. The approximate version of the APX algorithm terminates as soon as RANGE_COUNT returns a number within $[(\phi - \epsilon)N, (\phi + \epsilon)N]$. Specifically, the condition at Line 10 of Figure 6 is changed to $|cnt - \phi N| > \epsilon N$. For BAB (see Figure 8), we need to change the termination conditions at Lines 10 and 16, such that the algorithm stops when there exists a leaf entry in S (a set of temporary entries to be processed) whose range of possible ranks is enclosed by $[(\phi - \epsilon)N, (\phi + \epsilon)N]$. Thus, we replace the condition of Line 10 by "$cur_l + 1 \geq (\phi - \epsilon)N \ \wedge \ cur_l + \omega_l(e_l) \leq (\phi + \epsilon)N \ \wedge \ e_l$ is a leaf entry". The first condition ensures that all elements with ranks lower than $(\phi - \epsilon)N$ have been pruned, while the second condition ensures that the maximum possible rank of e_l does not exceed $(\phi + \epsilon)N$. Similarly, we replace the condition at Line 16 by "$N - cur_u \leq (\phi + \epsilon)N \ \wedge \ N + 1 - (cur_u + \omega_u(e_u)) \geq (\phi - \epsilon)N \ \wedge \ e_u$ is a leaf entry".

All three quantile algorithms can be adapted to generate progressively more refined estimates before the exact result is found. The progressive versions of the algorithms are identical to the methods described in Section 4, with only one difference. When the termination condition is checked, we compute and output the minimum value of ϵ (if any) for terminating, had the algorithm been approximate.

5.2 Batch quantile queries

A batch quantile query retrieves a set of quantiles $\phi_1, \phi_2, \cdots, \phi_m$ from the database, where $\phi_1 < \phi_2 < \cdots < \phi_m$ and each ϕ_i is a real value in the range $[0, 1]$. Batch quantiles offer a sketch of the underlying F value distribution. A naive solution would process individual quantile queries separately. We aim at reducing the total cost by exploiting intermediate results from previous computation.

INC can directly be applied for a batch query. During search, an element is reported if its rank is exactly $\phi_i N$ for some $i \in m$. The algorithm terminates until the $\phi_m N$-th element is found. For APX, the first quantile (i.e. ϕ_1) is computed as usual. In addition, we maintain a set C for storing computed intermediate coordinates (λ, cnt) (see Figure 6). These coordinates can be exploited to reduce the initial search space of the algorithm and improve the overall performance. Before computing the second quantile

(i.e. ϕ_2), the initial pair (λ_l, cnt_l) is replaced by the pair in S with the maximum count not greater than $\phi_2 N$. Similarly, the pair (λ_u, cnt_u) is replaced by the pair in S with the minimum count not smaller than $\phi_2 N$. Similarly, intermediate coordinates computed in this round are added to C for reducing the search space of the next quantile.

We can also define an efficient version of BAB for batch quantile queries. We compute the quantiles in ascending order of their ϕ values. During the computation of the first quantile (i.e. ϕ_1), any entry pruned on the upper side (i.e. at Lines 12–14 of Figure 8) is added to another set S'. After the first quantile is computed, we need not start the computation of the second quantile (i.e. ϕ_2) from scratch. We simply reuse the content of S in the last computation. Moreover, temporarily pruned entries in S' are moved back to S. Finally, we initialize cur_u to 0, reuse the previous value of cur_l, and begin the algorithm at Line 5 in Figure 8. The same procedure is repeated for subsequent quantiles. In this way, no tree nodes are accessed more than once.

5.3 Quantile computation without aR-trees

R-trees are commonly used for indexing spatial data in GIS or as multi-attribute indexes in DBMS, in general. On the other hand, aR-trees are not as popular, as they are mainly used for aggregate queries. For the case where only an R-tree is available on a spatial dataset (not an aR-tree), we can still apply the BAB algorithm. For each non-leaf entry, we compute the *expected* the number of objects (i.e., the aggregate value) and use this value instead of the actual count. A rough estimate can be derived from the level of the entry in the tree and the average R-tree node occupancy (which is a commonly maintained statistic). Naturally, BAB will not compute exact quantiles in this case, but values which are hopefully close to the exact result. In Section 6, we experimentally evaluate the accuracy of BAB on R-trees, by comparing its results with the exact quantiles.

In some streaming applications (e.g., traffic monitoring, mobile services), spatial data could not be indexed effectively due to high update rates and/or constrained storage space. In such cases, a common approach is to maintain spatial histograms [22] for approximate query processing. A spatial histogram consists of a set of entries, each associated with a MBR and the number of points inside it. We can derive an approximate quantile from such histograms, by applying a single pass of the BAB algorithm to prune histogram entries e that definitely do not containing the quantile. Based on the remaining (non-pruned) entries, we then compute the approximation and its respective error ϵ (in terms of rank).

6 Experimental Evaluation

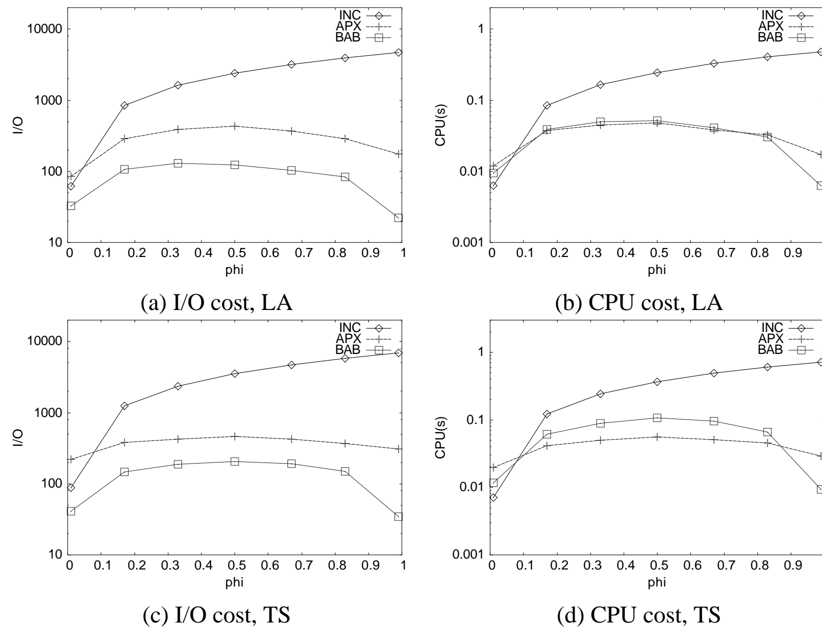
In this section, we evaluate the proposed algorithms using synthetic and real datasets. Uniform synthetic datasets were generated by assigning random numbers to dimensional values of objects independently. The default cardinality and dimensionality of a synthetic dataset are $N = 200K$ and $d = 2$. We also used real 2D spatial datasets from Tiger/Line[1], LA (131K points) and TS (194K points). Attribute values of all datasets are normalized to the range $[0, 10000]$. Each dataset is indexed by a `COUNT` aR-tree [17] with disk page size of 1K bytes.

[1] www.census.gov/geo/www/tiger/

Unless otherwise stated, the default searched quantile is $\phi = 0.5$ (i.e., median) and the ranking function F is defined as the Euclidean distance from a given query point, which follows the distribution of the dataset. All algorithms (INC for incremental search, APX for iterative approximation, BAB for branch-and-bound quantile) were implemented in C++. We also experimented with BAB-, a computationally cheaper variant of BAB that attempts pruning using only entries with upper bounds smaller than the quantile (i.e. Lines 12–17 in Figure 8 are not executed). All experiments were performed on a Pentium IV 2.3GHz PC with 512MB memory. The I/O cost corresponds to the number of aR-tree nodes accessed. A memory buffer of size 5% the number of aR-tree nodes is used by APX to avoid excessive number of page faults at repetitive RANGE_COUNT queries. For each experimental instance, the query cost is averaged over 100 queries with the same properties.

6.1 Experimental Results

Figure 10 shows the cost of the algorithms on real datasets LA and TS, as a function of ϕ. The results verify the analysis in Section 4.4 that APX and BAB have higher cost in computing the median than in calculating extreme quantiles (with ϕ close to 0 or 1). The cost of INC grows linearly with ϕ. APX is more efficient because it is based on RANGE_COUNT functions which can be answered effectively by the aR-tree. BAB incurs the lowest I/O overhead, indicating that the branch-and-bound approach only needs to explore a small fraction of the index. However, the CPU cost of BAB is slightly higher than APX because BAB requires considerable time computing the ω values of the intermediate entries. In practice, the I/O cost dominates, and thus BAB is by far the most efficient method.

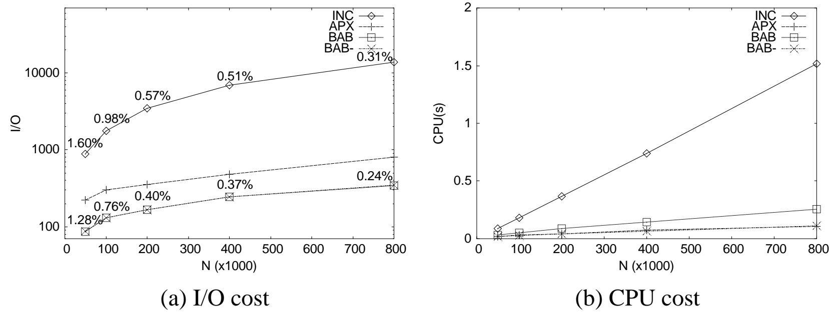


(a) I/O cost, LA (b) CPU cost, LA

(c) I/O cost, TS (d) CPU cost, TS

Fig. 10. Cost as a function of ϕ

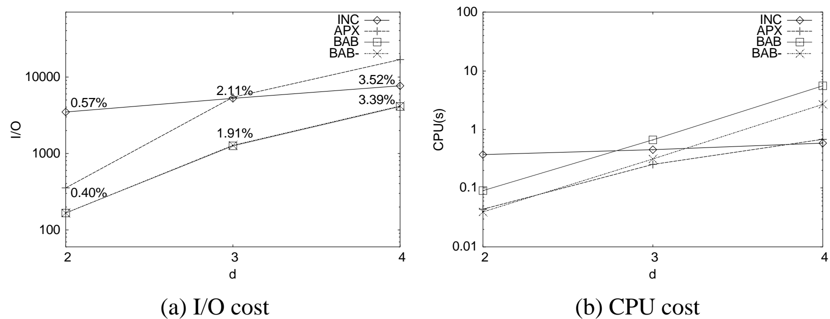
Next, we study the effect of database size N (Figure 11). As expected, BAB has the lowest I/O cost and outperforms the other two algorithms. BAB- has little I/O overhead over BAB, although BAB- misses opportunities for pruning entries with rankings higher than the quantile. In addition, BAB- has the lowest CPU cost among all algorithms. The numbers over the instances of INC and BAB are the maximum memory requirements of the algorithms (for storing the heap and set S) as a percentage of aR-tree size. Note that they are very low and decrease with N.



(a) I/O cost (b) CPU cost

Fig. 11. Cost as a function of N, Uniform data, $d = 2$, $\phi = 0.5$

Figure 12 shows the cost of the algorithms on uniform data as a function of dimensionality d (together with the maximum memory requirements of INC and BAB). BAB remains the least expensive algorithm in terms of I/O. APX has the highest I/O cost at $d = 4$ because RANGE_COUNT becomes less efficient at higher dimensionality. On the other hand, the CPU costs of BAB and BAB- increase significantly with dimensionality. As d increases, the F-intervals of the entries become wider and intersect many F-intervals of other entries. INC and APX do not spend time on pruning entries so they incur high I/O cost. On the other hand, BAB eliminates disqualified entries carefully at the expense of higher CPU cost. Note that memory requirements of the algorithms remain within acceptable bounds, even at $d = 4$.

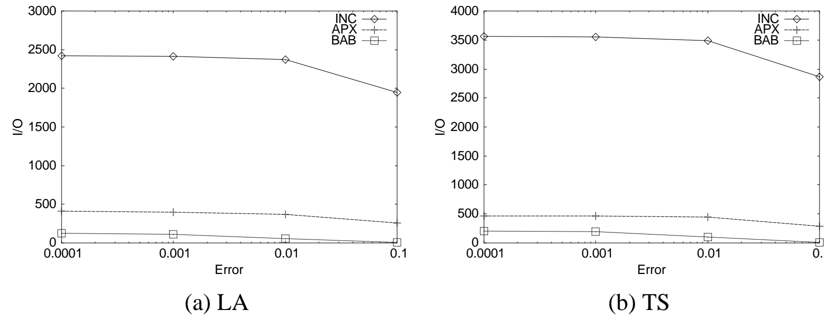


(a) I/O cost (b) CPU cost

Fig. 12. Cost as a function of d, Uniform data, $N = 200K$, $\phi = 0.5$

We also compared the efficiency of our algorithms in retrieving approximate quantiles. Recall that an ϵ-approximate quantile has a rank within $[(\phi - \epsilon)N, (\phi + \epsilon)N]$, where N is the data size. Figure 13 compares the algorithms for various values of ϵ. As expected, all costs decrease with the increase of ϵ, however, not at the same rate.

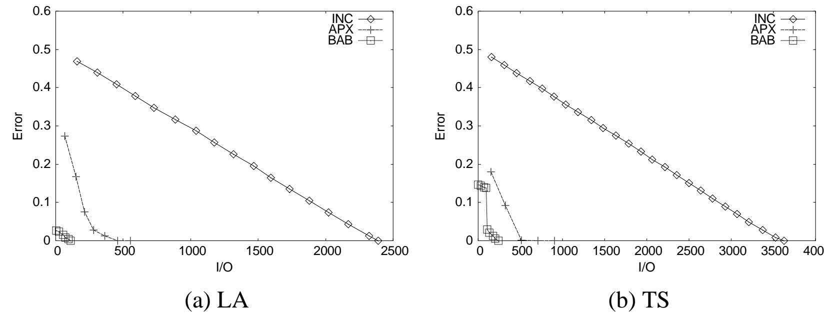
The cost of BAB decreases by 95% when ϵ changes from 0.0001 to 0.1, while the corresponding rates for APX and INC are 40% and 20%, respectively. This implies that BAB is not only the best algorithm for exact quantiles, but also has the highest performance savings for approximate computation.



(a) LA (b) TS

Fig. 13. I/O cost as a function of ϵ, $\phi = 0.5$

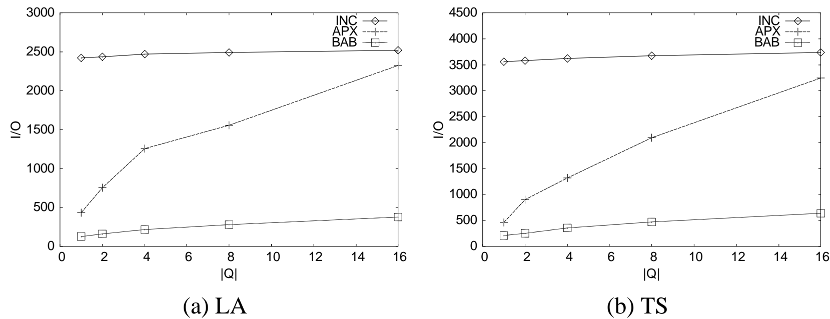
Figure 14 shows the progressiveness of the algorithms for a typical quantile query. As more disk page are accessed, all the algorithms continuously refine their intermediate results with decreasing error ϵ. The values at $\epsilon = 0$ corresponds to the case when the exact answer is found. BAB is the most progressive algorithm, followed by APX and then INC. Thus, BAB provides informative results to users, very early and way before it converges to the exact result.



(a) LA (b) TS

Fig. 14. Progressiveness of the algorithms for a typical query, $\phi = 0.5$

We also investigated the cost of extracting quantiles based on nearest-site distance ranking; a problem discussed in Section 3. Given a set $Q = \{q_1, q_2, \ldots, q_m\}$ of query points (sites), the ranking of a data point p is defined by $F(p) = \min_{i\in[1,m]} dist(q_i, p)$, where $dist$ denotes the Euclidean distance between two points. In the experiment, query points follow the distribution of data points. Figure 15 shows the cost of the algorithms as a function of the number of query points $|Q|$. As $|Q|$ increases, F-intervals of more entries overlap and the costs of all algorithms increase. The cost of BAB increases sublinearly with $|Q|$ and the algorithm outperforms its competitors by far.



(a) LA (b) TS

Fig. 15. I/O cost as a function of $|Q|$, $\phi = 0.5$

Finally, we study the accuracy of the BAB algorithm on regular R-trees (i.e., not aR-trees), where the count of each entry is estimated from its level, as discussed in Section 5.3. Figure 16 shows the observed error of BAB as a function of ϕ, on both real datasets LA and TS, indicating the difference of ranks (as a fraction of N) between the result and the actual quantile. Our solution produces fairly accurate results for real datasets indexed by regular R-trees. The maximum observed errors for LA and TS are just 0.07 and 0.05, respectively. The error is maximized at the median but it becomes negligible for extreme quantiles. As discussed, the I/O cost of BAB is maximized at $\phi = 0.5$. Higher I/O cost leads to higher error because counts of more entries need to be estimated.

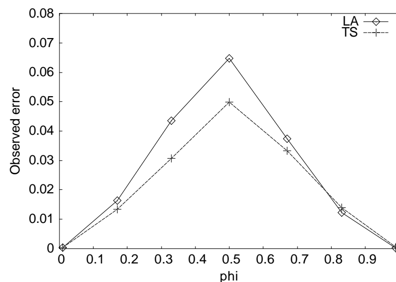


Fig. 16. Observed error of BAB as a function of ϕ, on regular R-trees

7 Conclusion

We identified the problem of computing dynamically derived quantiles in multidimensional datasets. We proposed three quantile algorithms (INC, APX, and BAB) which operate on aggregate R-trees. Also, we analyzed their performance qualitatively and suggested solutions for handling variants of quantile queries. INC is very expensive for high values of ϕ (i.e., quantile value). Although the cost of APX is relatively insensitive to ϕ, the algorithm accesses the aR-tree multiple times. BAB is the best algorithm as it traverses the tree carefully, pruning unnecessary nodes, and minimizing I/O cost.

In this paper we assume relatively low dimensionality, where aR-trees are effective. For high-dimensional spaces, however, the efficiency of aR-tree (as well as other space-partitioning access methods) drops significantly, in which case the algorithms may require accessing the entire dataset. Quantile computation in these environments is an interesting topic for future work.

References

1. K. Alsabti, S. Ranka, and V. Singh. A One-Pass Algorithm for Accurately Estimating Quantiles for Disk-Resident Data. In *VLDB*, 1997.
2. Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Sampling Algorithms: Lower Bounds and Applications. In *STOC*, 2001.
3. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time Bounds for Selection. *J. Comput. Syst. Sci.*, 7:448–461, 1973.
4. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD*, 1993.
5. K. Clarkson, D. Eppstein, G. Miller, C. Sturtivant, and S.-H. Teng. Approximating Center Points with Iterated Radon Points. *Int. J. Comp. Geom. and Appl.*, 6(3):357–377, 1996.
6. G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective Computation of Biased Quantiles over Data Streams. In *ICDE*, 2005.
7. R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.
8. A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to Summarize the Universe: Dynamic Maintenance of Quantiles. In *VLDB*, 2002.
9. M. Greenwald and S. Khanna. Space-Efficient Online Computation of Quantile Summaries. In *SIGMOD*, 2001.
10. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.
11. G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *TODS*, 24(2):265–318, 1999.
12. S. Jadhav and A. Mukhopadhyay. Computing a Centerpoint of a Finite Planar Set of Points in Linear Time. In *ACM Symposium on Computational Geometry*, 1993.
13. I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, 2001.
14. G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate Medians and other Quantiles in One Pass and with Limited Memory. In *SIGMOD*, 1998.
15. G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *SIGMOD*, 1999.
16. J. I. Munro and M. Paterson. Selection and Sorting with Limited Storage. *Theor. Comput. Sci.*, 12:315–323, 1980.
17. D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *SSTD*, 2001.
18. D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, 2003.
19. M. Paterson. Progress in selection. *Technical Report, University of Warwick, Conventry, UK*, 1997.
20. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.
21. I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *VLDB*, 2001.
22. N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic Multidimensional Histograms. In *SIGMOD*, 2002.