Evaluating Mid-(k, n) Queries Using B^+-Tree*

Dongseop Kwon, Taewon Lee, and Sukho Lee

School of Electrical Engineering and Computer Science,
Seoul National University, Seoul 151-742, Korea
{dongseop, taewon}@gmail.com, shlee@snu.ac.kr

Abstract. Traditional database systems assume that clients always consume the results of queries from the beginning. In various new applications especially in WWW, however, clients frequently need a small part of the result from the middle, e.g. retrieving a page in a bulletin board in WWW. To process this partial retrieval, traditional database systems should find all the records and discard unnecessary ones. Although several algorithms for top-k queries have been proposed, there has been no research effort for partial retrieving from the middle of an ordered result. In this paper, we define a mid-(k,n) query, which retrieves n records from the k^{th} record of an ordered result. We also propose an efficient algorithm for mid-(k,n) queries using a slightly modified B^+-Tree, named the B^{+^c}-Tree. We provide the theoretical analysis and the experimental results that the proposed technique evaluates mid-(k,n) queries efficiently.

1 Introduction

In various new applications such as WWW, the results of users' queries are generally huge. It is because users in these applications do not prefer to specify appropriate predicates or they just want to look over all the data to find useful information.

For example, a lot of web sites provide online bulletin boards or archives of articles. In many cases, those bulletin boards are so huge that they have millions of articles which have been archived for years. Since they cannot display all articles in one web page, they display only several of the articles as a page, and provide links to access other pages. With these links, users can directly access any page they want. From the viewpoint of a server, all pages are randomly requested because there are numerous requests from users simultaneously and the WWW uses a connectionless protocol. Therefore, retrieving the k^{th} record efficiently becomes important and essential especially in the WWW environments. Naïve or tricky solutions are commonly used for this problem at present, such as retrieving all and skipping unnecessary part, or using complicated subqueries.

Although there are several works on top-k queries, there has been no research effort for this partial retrieval from the middle of an ordered result, as far as we

* This work was supported in part by the Brain Korea 21 Project and in part by the Ministry of Information & Communications, Korea, under the Information Technology Research Center (ITRC) Support Program in 2005.

know. In this paper, we define a *mid-(k, n) query* as a query for retrieving n records from the k^{th} record of an ordered result. In addition, we propose an efficient processing algorithm for mid-(k, n) queries using a slightly modified B^{+}-Tree[1], named the B^{+c}-Tree. Each pointer of an internal node of a B^{+c}-Tree keeps the number of leaf records in its subtree. Using this additional information, the B^{+c}-Tree evaluates mid-(k, n) queries efficiently. We present the theoretical analysis of the cost of the B^{+c}-Tree and the experimental results that the proposed technique outperforms the B^{+}-Tree.

The rest of this paper is organized as follows: In Section 2, we review related work. Section 3 defines mid-(k, n) queries and naïve solutions. In Section 4, we propose the B^{+c}-Tree for the efficient processing of mid-(k, n) queries. We analyze the cost of the B^{+c}-Tree in Section 5. Section 6 presents the experimental results to compare our technique with the B^{+}-Tree. Finally, Section 7 concludes the paper.

2 Related Work

As far as we know, there has been no research effort for mid-(k, n) queries. Top-k queries and quantile queries are possible candidates that can be available for processing mid-(k, n) queries. In this section, we review evaluation techniques for these two types of queries, and present the problems of these techniques for using mid-(k, n) queries.

2.1 Top-k Queries

There are several research work for top-k queries. Carey and Kossmann [2] present a method to limit the cardinality of a query result by adding the 'STOP AFTER' clause to a simple SQL, and propose efficient processing strategies for 'STOP AFTER' queries. Donjerkovic and Ramakrishnan [3] propose a probabilistic approach to optimize the top-k query processing. Chaudhuri and Gravano [4] propose a technique that translate a top-k query into a single range query using multi-dimensional histograms. Chen and Ling [5] propose a sampling-based method to translate a top-k query to a range query. However, all these techniques cannot be adopted for mid-(k, n) queries directly. The only naïve way is retrieving all the result of a top-k query which includes all the result of a mid-(k, n) query, and skipping all the unnecessary records from the beginning, which is greatly inefficient unless k is very small.

2.2 Quantile Queries

Theoretically, A quantile query, which is the problem of selecting selecting the i^{th} order statistic from N elements, can be solved in $O(N)$ time bound in average case [6]. There are also several research works [7,8] for quantile queries for database systems. However, quantile queries are different from general mid-(k, n) queries, because mid-(k, n) queries have to retrieve a set of records from the k^{th} to the $(k + n - 1)^{th}$ instead of retrieving the k^{th} record only. Of course, the algorithms for quantile queries can be used for mid-(k, n) queries with a little extension. However, the algorithms for quantile queries need to read all data at least once, since they are designed for the case of no-index. Therefore, they are not efficient for large volumes of data. Our proposed algorithm can evaluate mid-(k, n) queries more efficiently because it uses an index on the data.

3 Mid-(k, n) Queries

We define a **mid-(k, n) query** as a query that retrieves n records from the k^{th}. The semantic of mid-(k, n) queries can be expressed by using the 'LIMIT n OFFSET k' clause, which are supported in PostgreSQL 7.4.7.

For example, the following query is for accessing *pageNo* page directly in an online bulletin board, where one page has *pageSize* articles.

```
SELECT * FROM BULLETIN1 ORDER BY wdate DESC
LIMIT pageSize OFFSET (pageNo-1) × pageSize
```

A naïve way to process mid-(k, n) queries is to sort all the records and to skip unnecessary records from the beginning until the k^{th}. One possible alternative way is that a system executes a top-($k + n$) query instead of a mid-(k,n) query and discards the first $k - 1$ records. However, it is greatly inefficient unless k and n are small. In general cases, there is no efficient way to process mid-(k, n) queries.

In this paper, therefore, we propose an efficient algorithm for mid-(k, n) queries with some restrictions as follows: (1) There is an index on the columns which are used in the 'ORDER BY' clause. (2) Only the columns used in the 'ORDER BY' clause can appear in the 'WHERE' clause. As we mentioned in Section 1, many web applications often use only one sorting order. For this case, the first restriction is quite reasonable as a way of tuning the performance. In addition, the second restriction is also very common because users do not specify any search predicates in most cases. Therefore, our approach is useful for various applications especially in WWW.

4 B^{+c}-Tree

In this section, we give the detailed description about our proposed technique, the B^{+c}-Tree. The main difference between the B^{+c}-Tree and the B$^+$-Tree is that the B^{+c}-Tree keeps an additional count information with each pointer in the internal nodes. In the original B$^+$-Tree, an internal node has m child pointers and $m - 1$ keys. In the B^{+c}-Tree, as depicted in Figure 1, each pointer has the number of records in the leaf nodes of its subtree. For example, since the first leaf node has 2 records, the pointer pointing to the leaf node has a record count of 2. The first pointer of the root node has a record count of 7 because its subtree has 7 records in its leaf nodes. This record count can be maintained during an insertion and a deletion of a record. From now on, we will use PTR, KEY and

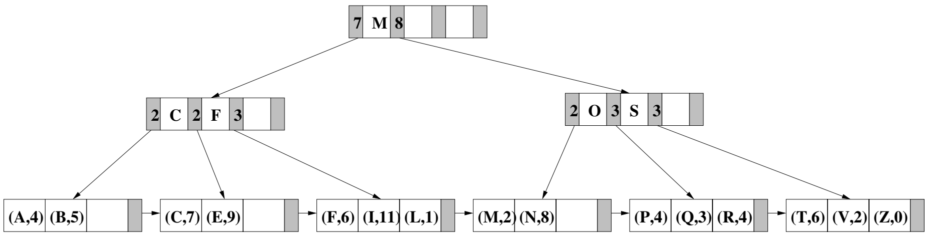


Fig. 1. An example of the B^{+c}-Tree

CNT to represent a pointer, a key and a record count, respectively. The range query is processed in the same way in both of the trees. By using the *CNT* values, the B^{+c}-Tree performs mid-(k, n) query more efficiently.

Now, we will present the algorithms for the mid-(k, n) query and the insertion and deletion of a record in the B^{+c}-Tree.

4.1 Algorithm for Mid-(k, n) Queries

For the convenience of the explanation, we divide mid-(k, n) queries into two classes.

– queries without search predicate
– queries with search predicates

For simplicity, we first describe two examples for each query class, and then present the detailed algorithm. Suppose there are the B^{+}-Tree and the B^{+c}-Tree with increasing order on the data records.

Example 1. Suppose a mid-(9,1) query is given. To find out the 9^{th} record from the ordered result, the B^{+}-Tree has to start to find from the first lead without any traversal from the root. Since the 9^{th} record is located in the 4^{th} leaf node, the B^{+}-Tree follows the pointers to a next leaf nodes until it reaches the 4^{th} leaf node. Therefore, it needs 4 disk accesses.

The B^{+c}-Tree, as depicted in Figure 1, can utilize the *CNT* values to find the 9^{th} record. By looking at the first entry of the root node, the B^{+c}-Tree can guess that there are 7 records in the first subtree of the root node. Therefore, there is no need to go to the first subtree. The B^{+c}-Tree follows the second pointer, and then examines the *CNT* values in that node. Since the *CNT* value for the first pointer is 2, the B^{+c}-Tree knows that the 9^{th} record is in the first child of the node. Therefore, after following the first pointer to get to the leaf, The B^{+c}-Tree can find the 9^{th} record. It requires 3 disk accesses. We can save 1 disk access in this case.

Example 2. Suppose there is a search predicate given with mid-(5,1) query. Let the search predicate as '`D`$\leq$ *key* $\leq$`T`'.

Since the query is to find the 5^{th} record with a range predicate, we cannot use the *CNT* values in the B^{+c}-Tree directly. We first should know the position

Algorithm 1: Mid-(k, n) query

```
Function Mid-(k, n) (range, start, limit)
begin
    $pos \leftarrow$ GetPosition $(range)$
    $Result \leftarrow \emptyset$, $S \leftarrow 0$, $N \leftarrow$ the root node of the B$^{+c}$-Tree
    while $N$ is an internal node do
        find the first entry $e_i$ that satisfies $S + \sum_{n=0}^{i} CNT_n >= start - pos$
        $S \leftarrow S + \sum_{n=0}^{i-1} CNT_n$
        $N \leftarrow$ child node pointed to by $PTR_i$
    endw
    $R \leftarrow (start - pos - S)^{th}$ record of $N$
    while $|Result| < limit$ and $R$ in range do
        $Result \leftarrow Result \cup R$
        $R \leftarrow$ next record
    endw
    return $Result$;
end
```

Algorithm 2: Getting the poisition of the lower bound predicate

```
Function GetPosition (range)
begin
    if range has lower bound then
        $R_f \leftarrow$ the smallest record in the range
        $pos \leftarrow$ position of $R_f$ in the ordered result of all the records
    else
        $pos \leftarrow 0$
    endif
    return $pos$
end
```

of D. During the original key lookup process in the B$^+$-Tree, the B^{+c}-Tree can calculate the number of records whose key is less than D. In this example, we can find out that there is 3 records whose key is less than D. Therefore, in this case, we have to find 8th record as the same way as in Example 1.

The algorithm for mid-(k, n) queries is described in detail in Algorithm 1. Note that if there is no search predicate or no lower bound in a search predicate, we can directly go to the record at the desired position like Example 1. If there is a search predicate, first we have to find the position of the lower bound of the predicate.

If there is a search predicate and it has lower bound, the B^{+c}-Tree always have to traverse the tree twice from the root to a leaf. On the contrast, the B$^+$-Tree should retrieve not only all nodes in a path from the root to a leaf, but also all the leaf nodes from the leaf to the leaf node where the k^{th} records is located.

Algorithm 2 is for finding the position of the lower bound record in this case.

Table 1. Notation

symbol	meaning
α	the average fill-factor of a node
h	the height of a tree
S_{page}, S_{header}, S_{record}, S_{key}, $S_{pointer}$, S_{count}	the size of a disk page, a header of a node, a record, a key (KEY), a pointer (PTR), and a record count (CNT), respectively
n_{record}	the number of records in a leaf node
M_{int}, M_{leaf}	the maximum number of pointers in an internal node, and records in a leaf node, respectively
C_{insert}, C_{delete}, C_{search}, C_{mid-k}	the number of disk access for an insert query (in case of no overflow), for a delete query (in case of no underflow), for an exact matching query, and for a mid-k query, respectively

4.2 Algorithm for Inserting and Deleting a Record

In this section, we describe the algorithms of the insertion and deletion for the B^{+c}-Tree.

To insert a record, we should choose a leaf node N where the search key value would appear. To keep correct CNT values, we should increase each CNT in the path from the root to N by one during the insertion. If a node is full, we should split it into two nodes. After the splitting, we should adjust the CNT value of each PTR in the parent node.

Deleting a record can be performed in the similar way of the insertion, except that we decrease each CNT values in the path from the root to the leaf where the deleted key exists by one. If we should merge two nodes, we can add up the CNT values of the pointers to the two nodes being merged.

5 Analysis

We now analyze the cost of the B$^+$-Tree and the B^{+c}-Tree in terms of the number of disk accesses. The notation used in this section is summarized in Table 1.

5.1 Cost of the B$^+$-Tree

Lemma 1. *The maximum cardinalities of an internal node and a leaf node of a B$^+$-Tree are as follows:*

$$M_{int} = \lfloor \frac{(S_{page} - S_{header} + S_{key})}{(S_{key} + S_{pointer})} \rfloor$$

$$M_{leaf} = \lfloor \frac{(S_{page} - S_{header} - S_{pointer})}{S_{record}} \rfloor$$

Proof. An internal node in a B$^+$-Tree consists of a header, M_{int} pointers, and $(M_{int} - 1)$ keys. If a node is to be stored in a disk page, the size of a node cannot be bigger than the size of a disk page. Therefore $S_{header} + (M_{int} -$

$1) \cdot S_{key} + M_{int} \cdot S_{pointer} \leq S_{page}$. From this, $M_{int} \leq \frac{(S_{page} - S_{header} + S_{key})}{(S_{key} + S_{pointer})}$. Since M_{int} is the maximum integer value that satisfy the previous equation, $M_{int} = \lfloor \frac{(S_{page} - S_{header} + S_{key})}{(S_{key} + S_{pointer})} \rfloor$.

A leaf node consists of a header, M_{leaf} records and a pointer to the following leaf node. Therefore, as the same manner, M_{leaf} is $\lfloor \frac{(S_{page} - S_{header} - S_{pointer})}{S_{record}} \rfloor$.

Lemma 2. *The height of a B$^+$-Tree is as follows:*

$$h = \log_{(\alpha \cdot M_{int})}(\lceil \frac{n_{record}}{\alpha \cdot M_{leaf}} \rceil)$$

Proof. If the average fill-factor of a node is α, a leaf node has the average $\alpha \cdot M_{leaf}$ data records. Therefore the B$^+$-Tree has $\lceil \frac{n_{record}}{\alpha \cdot M_{leaf}} \rceil$ leaf nodes. An internal node has the average $\alpha \cdot M_{int}$ pointers. Therefore, from these values, the height of a B$^+$-Tree is $\log_{(\alpha \cdot M_{int})}(\lceil \frac{n_{record}}{\alpha \cdot M_{leaf}} \rceil)$.

Theorem 1. *If there is no overflow during the insertion and no underflow during the deletion, cost of the B$^+$-Tree are as follows:*

$$C_{insert} = h + 1, \quad C_{delete} = h + 1, \quad C_{search} = h, \quad C_{mid-k} = h + \lceil \frac{k}{\alpha \cdot M_{leaf}} \rceil$$

Proof. To insert a record, we first find the leaf node that the inserted item can be stored. For this, it is necessary to read h disk pages. Then, it needs to 1 disk page for storing the item in the leaf. Therefore, C_{insert} is $h + 1$. C_{delete} is the same as C_{insert}. To process a exact matching query, it needs to read h disk pages from the root to a leaf. Therefore, C_{search} is h. Finally, to process a mid-(k, n) query, we first perform 1 exact matching query if a low bound is specified, and then read k records sequentially. Since k records is stored in $\lceil \frac{k}{\alpha \cdot M_{leaf}} \rceil$ disk pages, C_{mid-k} is $h + \lceil \frac{k}{\alpha \cdot M_{leaf}} \rceil$.

5.2 Cost of the B^{+c}-Tree

With the similar way to the B$^+$-Tree, cost of the B^{+c}-Tree is as follows:

Lemma 3. *The maximum cardinalities of an internal node and a leaf node of a B^{+c}-Tree are as follows:*

$$M_{int} = \lfloor \frac{(S_{page} - S_{header} + S_{key})}{(S_{key} + S_{pointer} + S_{count})} \rfloor$$
$$M_{leaf} = \lfloor \frac{(S_{page} - S_{header} - S_{pointer})}{S_{record}} \rfloor$$

Lemma 4. *The height of a B^{+c}-Tree is as follows:*

$$h = \log_{(\alpha \cdot M_{int})}(\lceil \frac{n_{record}}{\alpha \cdot M_{leaf}} \rceil)$$

Theorem 2. *If there is no overflow during the insertion and no underflow during the deletion, cost of the B^{+c}-Tree are as follows:*

$$C_{insert} = 2h, \quad C_{delete} = 2h, \quad C_{search} = h, \quad C_{mid-k} = 2h$$

Proof. For the insertion, the B^{+c}-Tree finds a leaf node and writes the inserted record into the leaf, same as the B^{+}-Tree. Then, since the counters in the internal nodes from the leaf to the root should be adjusted, it needs additional $h-1$ disk writes. Therefore, C_{insert} is $(h+1)+(h-1) = 2h$. C_{delete} is the same as C_{insert}. The algorithm for exact matching queries of the B^{+c}-Tree is the same as that of the B^{+}-Tree. Therefore, C_{search} is h. Finally, to process a mid-(k, n) query, one traversal from the root to a leaf as same as the B^{+}-Tree is required if a lower bound is specified. Then, we need one more traversal from the root to a leaf in order to find the k^{th} records. Therefore, C_{mid-k} is $h+h = 2h$.

Note that the buffering effect of a disk cache is ignored in this analysis. If we ignore the buffering effect, we can say that the performance of the B^{+}-Tree for a mid-(k, n) query can be better than that of the B^{+c}-Tree when $\lceil \frac{k}{\alpha \cdot M_{leaf}} \rceil < h$, which means k is very small. With disk caches, however, the cost of the second traversal is ignorable because most of the accessed disk pages in the second traversal are the same as those in the first traversal. Therefore, the cost of the B^{+c}-Tree for mid-(k, n) queries is almost same as that of the B^{+}-Tree even in case of small k, like the experimental result shown in Section 6. For the same reason, the cost of insertions or deletions of the B^{+c}-Tree is almost same as that of the B^{+}-Tree when disk caches are used.

6 Experiments

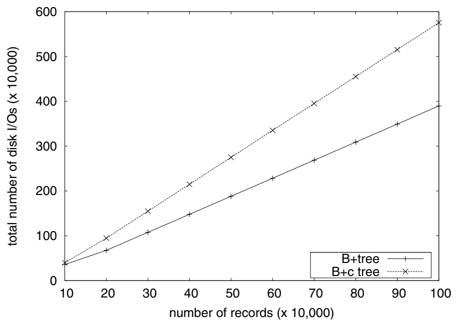
In this section, we present the result of an experimental study to show the validity and the effectiveness of our approach. We have implemented a disk based B^{+}-Tree and B^{+c}-Tree on a 1GHz linux machine with 768MB main memory.

In the implementation, we directly managed the LRU buffer. We used 4KB disk pages for buffer cache. Each key and pointer occupy 4 bytes respectively. *CNT* size is 4 bytes. Therefore in the B^{+}-Tree, an internal node can contain about 510 pointers and in the B^{+c}-Tree, it can contain about 340 pointers.

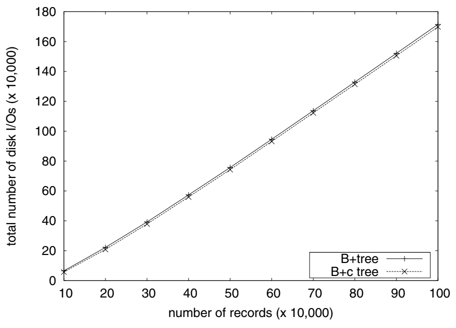
Since the distribution of data does not affect our experiment, we used uniformly distributed data for the key values. We generated 1,000,000 data records for both of the trees. From now on, if there is no mention on the buffer cache size, we used 100 4KB-buffer pages which is about 5% of the total nodes in B^{+}-Tree.

6.1 Overheads of the B^{+c}-Tree

In this experiment, we inserted varying number of data records into both trees and compared the total disk I/Os. As shown in Figure 2(a), without the buffer cache, the B^{+c}-Tree requires about 1.5 times more disk accesses than B^{+}-Tree. However, as shown in Figure 2(b), if the buffer cache is used, there is almost

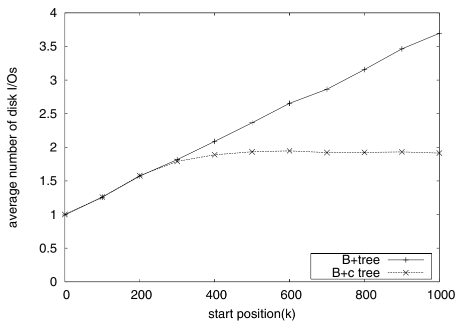


(a) without the buffer cache

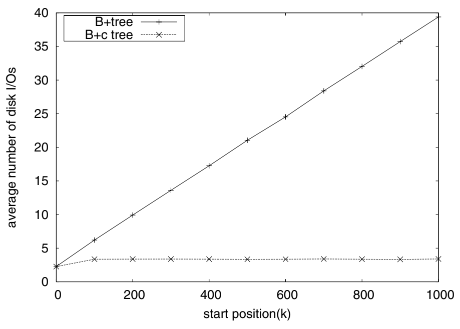


(b) with the buffer cache

Fig. 2. Insertion cost with varying number of records



(a) non-clustered index



(b) clustered index

Fig. 3. Mid-(k, n) query cost with varying k

no difference between the two trees. It is because top level nodes are almost always in the cache which decreases the overhead of the B^{+c}-Tree as described in Section 5. Like the result of the insertion cost with the buffer cache, there is also no difference between the range query costs of the two trees. From these experimental results, we showed that the overhead of the B^{+c}-Tree is ignorable in spite of the additional record counts with the buffer cache.

6.2 Cost for Mid-(k, n) Queries

In this experiment, we performed mid-(k, 20) queries with a non-clustered index and a clustered index. The reason why we used a clustered index here is that it is common to use a strong-clustered index in real applications as mentioned in Section 1. In the non-clustered index used in this experiment, a leaf node has 4-byte keys and 4-byte pointers. For the clustered index, we assumed a leaf node has a number of 4-byte keys and 100-byte sized tuples. We varied k from 0 to 1,000 in both experiments.

In Figure 3(a), the performance gap between the B$^+$-Tree and the B^{+c}-Tree grows larger from the point where k is about 300. Note that 300 is very small compared to the total number of records in the tree, which is 1,000,000 in this experiment. In the clustered index as in Figure 3(b), the performance of the B^{+c}-Tree is much better from the beginning. Since the B^{+c}-Tree is not affected by k as shown in Figure 3(b), the proposed technique is highly scalable so that it is appropriate for WWW applications.

Due to the buffering effect, the performance of the B^{+c}-Tree is almost same as that of the B^{+}-Tree in small k, as mentioned in Section 5.

7 Conclusion

Various new applications of database systems such as WWW have brought new needs of different kinds of queries which were not taken any notice in traditional database systems. One of those new kinds of queries is the mid-(k,n) query, which retrieves n records from the k^{th} record of an ordered result. In this paper, we have addressed the problem of mid-(k, n) queries and have proposed an efficient algorithm for processing mid-(k, n) queries using a slightly modified B^{+}-Tree, named the B^{+c}-Tree. We also presented the theoretical analysis of the cost of the proposed method and provided experimental evidence that our approach outperforms the B^{+}-Tree. Future work includes extending this technique for supporting more complex cases of mid-(k, n) queries with several predicates.

References

1. Comer, D.: Ubiquitous b-tree. ACM Computing Surveys **11** (1979) 121–137
2. Carey, M.J., Kossmann, D.: Reducing the braking distance of an sql query engine. In: Proceedings of 24th Int'l. Conf. on Very Large Data Bases. (1998) 158–169
3. Donjerkovic, D., Ramakrishnan, R.: Probabilistic optimization of top n queries. In: Proceedings of 25th Int'l. Conf. on Very Large Data Bases. (1999) 411–422
4. Chaudhuri, S., Gravano, L.: Evaluating top-k selection queries. In: Proceedings of 25th Int'l. Conf. on Very Large Data Bases. (1999) 397–410
5. Chen, C.M., Ling, Y.: A sampling-based estimator for top-k query. In: Proceedings of the 18th Int'l. Conf. on Data Engineering. (2002) 617–627
6. Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. Journal of Computer and System Sciences **7** (1973) 448–461
7. Alsabti, K., Ranka, S., Singh, V.: A one-pass algorithm for accurately estimating quantiles for disk-resident data. In: Proceedings of 23rd Int'l. Conf. on Very Large Data Bases. (1997) 346–355
8. Manku, G.S., Rajagopalan, S., Lindsay, B.G.: Approximate medians and other quantiles in one pass and with limited memory. In: Proceedings of the 1998 ACM SIGMOD Int'l. Conf. on Management of Data. (1998) 426–435