IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 16, Issue 3, Ver. VII (May-Jun. 2014), PP 54-60
www.iosrjournals.org

Processing of Top-k Selection Queries in Relational Database System

Neha Singh, P.K. Pandey*
Department of Computer Science, A.P.S. University, Rewa (M.P.)
**OSD, Additional Directorate Higher Education, Division Rewa (M.P.)-India*

Abstract: *In many applications, users specify target values for certain attributes, without requiring exact matches to these values in return. Instead, the result to such queries is typically a rank of the top-k tuples that best match the given attribute values. Focusing on the top-k items according to a ranking criterion constitutes an important functionality in many different query answering scenarios. The idea is to read only the necessary information mostly from secondary storage with the ultimate goal to achieve low latency. In this paper, we study the advantages and limitations of processing a top-k query by translating it into a single range query that traditional relational database management systems can process efficiently. In this work, we also consider processing such top-k queries under the constraint that the result items are members of a specific set, which is provided at query time it is also known as set-defined selection criterion.*
Keywords: *top-k query processing, index partitioning, mapping strategy, index partitioning*

I. Introduction

Internet Search engines rank the objects in the results of selection queries according to how well these objects match the original selection condition. For such engines, query results are not flat sets of objects that match a given condition. Instead, query results are ranked starting from the top object for the query at hand. Given a query consisting of a set of words, a search engine returns the matching documents sorted according to how well they match the query. For decades, the information retrieval field has studied how to rank text documents for a query both efficiently and effectively [1]. In contrast, much less attention has been devoted to supporting such top-k queries over relational databases. As the following example illustrates, top-k queries arise naturally in many applications where the data is exact, as in a traditional relational database, but where users are flexible and willing to accept non-exact matches that are close to their specification. The answer to such a query is a ranked set of the k tuples in the database that best match the selection condition.
Example: Consider a real-estate database that maintains information like the Price and Number of Bedrooms of each house that is available for sale. Suppose that a potential customer is interested in houses with four bedrooms, and with a price tag of around 3000 Rs. The database system should then rank the available houses according to how well they match the given user preference, and return the top houses for the user to inspect. If no houses match the query specification exactly, the system might return a house with, say, five bedrooms and a price tag close to 3000 Rs. as the top house for the query.

Unfortunately, despite the conceptual simplicity of top-k queries and the expected performance payoff, they are not yet supported by today's relational database systems. This support would free applications and end-users from having to add this functionality in their client code. To provide such support efficiently, we need processing techniques that do not involve full sequential scans of the underlying relations. The challenge in providing this functionality is that the database system needs to handle efficiently top-k queries for a wide variety of scoring functions. In effect, these scoring functions might change by user, and they might also vary by application, or by database.

It is also important that we are able to process such top-k queries with as few extensions to existing query engines as possible, since today's relational systems are significantly complex and performance sensitive. As in the case of processing traditional selection queries, one must consider the problem of execution as well as optimization of top-k queries. We assume that the execution engine is a traditional relational engine that supports single as well as possibly multidimensional indexes. Therefore, the key challenge is to augment the optimization phase such that top-k selection queries may be compiled into an execution plan that can leverage the existing data structures (i.e., indexes) and statistics (e.g., histograms) that a database system maintains. Simply put, we need to develop new techniques that make it possible to map a top-k query into a traditional selection query. It is also important that any such technique preserves the following two properties: (1) it handles a variety of scoring functions for computing the top-k tuples for a query, and (2) it guarantees that there are no false dismissals.

In this paper, we undertake a comprehensive study of the problem of mapping top-k queries into execution plans that use traditional selection queries. In particular, we use the database histograms to map a top-k query to a suitable range that encapsulates k best matches for the query. In particular, we study the sensitivity of the mapping algorithms to the following parameters: types of histograms available and their memory budgets, scoring functions, data distribution, and number of query attributes.

In this work, we also consider set-defined selections, where the items of interest are by whatever means identified upfront and represented in the query as a set of item ids. In addition, the query specifies the ranking attribute of interest and the result set size referred to as parameter k. The size of the selection set drastically inuences the design of an ideal index: When the selection set contains a small number of ids, the query is efficiently answered using
an index on the id attribute. In case the selection set contains most of the ids, the best performance is reached by reading the ids from the score-sorted lists.

II. Related Work

Our main focus is on exploring opportunities and limitations of efficiently mapping top-k queries into traditional relational queries. Carey and Kossman [2, 3] presented techniques to optimize queries that require only top-k matches. Their technique leverages the fact that when k is relatively small compared to the size of the relation, specialized sorting (or indexing) techniques that can produce the first few values efficiently should be used. Only after evaluating the score for each object are we able to use the techniques in [2, 4]. Hence, these strategies require a pre-processing step to compute the scoring function itself involving one sequential scan of all the data. In [5, 6], Fagin addresses the problem of finding top-k matches for a user query *q* involving several multimedia attributes. Each of these attributes is assumed to have a native sub-system that answers top-k queries involving only the corresponding attribute. The state-of-the-art algorithms (e.g., [7]) follow a multi-step approach. Their key step is identifying a set of points A such that p's k nearest neighbours are no further from p than a is, where a is the point in A that is furthest from p. References [8,9] study how to merge and reconcile top-*k* query results obtained from distributed databases when the databases use arbitrary, undisclosed scoring algorithms. There is a large body of existing work in the area of processing top-K queries, ranging from database systems [10,11]. Among the most prominent approaches are the so called threshold algorithms [5,12,13], which aggregate scores from score-sorted lists, while maintaining a threshold used for an early termination. Ranking join results based on aggregated scores obtained from multiple tables (i.e., top-K join processing) and embedding such ranking concepts in a query optimizer have
been addressed in [14,15]. Using query logs (i.e., historic workloads) with the aim of tuning a system's performance is encountered frequently [16, 17, 18, 19]. Applications where the workload is used to improve performance vary from index defragmentation [16], over cache replacement [18] to range queries [17]. Query logs have been used in [19] with the aim to reduce the number of distributed partitions by allocating tuples that are frequently used together to the same partition.

III. Query Model

In a traditional relational system, the answer to a selection query is a set of tuples. In contrast, the answer to a top-k query is an ordered set of tuples, where the ordering reflects how closely each tuple matches the given query. This section defines our query model precisely. Consider a relation R with attributes A1,….,An. A top-k query over R simply specifies target values for the attributes in R. Thus, a query is an assignment of values v1,….,vn to the attributes A1,……, An of R. In this paper, we will focus on top-k queries on continuous attributes (e.g., age, salary). Without loss of generality, we will also assume that the values of these attributes are normalized to be real numbers between 0 and 1.

Given a top-k query q, the database system with relation R uses some scoring function Score to determine how closely each tuple in R matches the target values v1,…….,vn specified in query q. Given a tuple t and a query q, we assume that Score(q, t) is a real number that ranges between 0 and 1. In this paper, we focus on three important scoring functions, namely Min, Euclidean, and Sum.

Definition: Consider a relation R = (A1,……, An).A1,…….,An are real-valued attributes ranging between 0 and 1. Then, given a query q = (q1,……., qn) and a tuple t = (t1,…., tn) from R, we define the score of t for q using any of the following three scoring functions:

$$\text{Min}(q, t) = \underset{i=1}{\overset{n}{\text{sum}}} \{1-|q_i-t_i|\}$$

$$\text{Euclidean}(q, t) = 1-\sqrt{\sum_{i=1}^{n} (q_i-t_i)^2/n}$$

$$\text{Sum}(q; t) = 1 - \sum_{i=1}^{n} |q_i - t_i| / n$$

A simple variation of the definition of the scoring functions above results from letting the different attributes have different weights. In general, the Min,Euclidean, and Sum functions that we use in this paper are just a few of many possible scoring functions. Our strategy for processing top-k queries can be adapted to handle a wide variety of such functions, as we will discuss. The key property that we ask from scoring functions is as follows:
Property 1: Monotonicity of Scoring Functions: Consider a relation R and a scoring function Score defined over it. Let q = (v_1,….., v_n) be a top-k query over R, and let t = (t_1,….., t_n) and t' = (t'_1,………., t'_n) be two tuples in R such that $|t'_i - q_i| \leq |t_i - q_i|$ for i = 1,….., n. (In other words, t' is at least as close to q as t for all attributes.) Then, Score(q, t') ≥ Score(q, t).

Intuitively, this property of scoring functions implies that if a tuple t' is closer, along each attribute, to the query values than some other tuple t is, then, the score that t' gets for the query cannot be worse than that of t. Fortunately, all interesting scoring functions that we could think of satisfy our monotonicity assumptions. In particular, the Euclidean, Min, and Sum scoring functions that we defined above satisfy this property. A possible SQL-like notation for expressing top-k queries is as follows [3]:

```
SELECT * FROM R
WHERE A1=v1 AND ... AND An=vn
ORDER k BY Score
```

The distinguishing feature of the query model is in the ORDER BY clause. This clause indicates that we are interested in only the k answers that best match the given WHERE clause, according to the Score function.

IV. Mapping a Top-k Query into a Traditional Selection Query

In this section we describe how to map a top-k query q into a relational selection query C_q that any traditional relational database management system can execute. Our goal is to obtain k tuples from relation R that are the best tuples for q according to a scoring function Score. Our query processing strategy consists of the following steps:
1. Use statistics on relation R to find a search score S_q.
2. Build a selection query C_q to retrieve all tuples in R with score Sq or higher for q.
3. Evaluate C_q over R.
4. Compute Score(q, t) for every tuple t in the answer for C_q.
5. If there are at least k tuples t in the result for C_q with Score(q,t) ≥ S_q, then output k tuples with the highest scores. Otherwise, choose a lower value for S_q and restart the process.

4.1 Choice of Search Score S_q: The key step for evaluating a top-k query q is determining score S_q: our algorithm retrieves all tuples t such that Score(q, t) ≥ S_q. If there are at least k such tuples, then our algorithm above succeeds in finding the top k matches for q. Otherwise, our choice of S_q is too high, and hence the query needs to be restarted with a lower value for S_q. Consequently, we should choose a value of S_q that is not too low, so that we do not retrieve too many candidate tuples from the database, but that is not too high either, so that we can obtain the top-k tuples without restarting the query. Our choice of S_q will be guided by the statistics that the query processor keeps about relation R.

In particular, we will assume that we have an n-dimensional histogram H that describes the distribution of values of R. We assume that H consists of a series of nonoverlapping buckets. Each bucket has associated with it an n-rectangle [a1,b1]x….x[an, bn], and stores the number of tuples in R that lie within the n-rectangle, together with other information. For efficiency, our choice of S_q will be based on histogram H, and not on the underlying relation R itself. More specifically, we choose S_q as follows:
a. Create (conceptually) a small, "synthetic" relation R', consistent with histogram H. R' has one distinct tuple for each bucket in H, with as many instances as the frequency of the corresponding bucket.
b. Compute Score(q, t) for every tuple t in R'.
c. Let T be the set of the top-k tuples in R' for q. Output $S_q = \min_{t \in T}$ Score(q, t).

We can conceptually build synthetic relation R' in many different ways. We will study two extreme query processing strategies that result from two possible definitions of R'.

The first query processing strategy, NoRestarts, results in a search score S_q that is low enough to guarantee that no restarts are ever needed as long as histograms are kept up to date. In other words, Step (5) above always finishes successfully, without ever having to reduce S_q and restart the process. For this, the NoRestarts strategy defines R' in a pessimistic way:

Given a histogram bucket b, the corresponding tuple t_b that represents b in R' will be as bad for query q as possible. More formally, t_b is a tuple in b's n-rectangle with the following property:

$$Score(q,t_b) = \min_{t \in T_b} Score(q, t)$$

where T_b is the set of all potential tuples in the n-rectangle associated with bucket b.

Lemma 1: Let q be a top-k query over a relation R. Let S_q be the search score computed by strategy NoRestarts for q. Then, there are at least k tuples t in R such that $Score(q,t) \geq S_q$.

The second query processing strategy, Restarts, results in a search score S_q that is highest among those search scores that might result in no restarts. This strategy defines R' in an optimistic way: given a histogram bucket b, the corresponding tuple t_b that represents t_b in R' will be as good for query q as possible. More formally, t_b is a tuple in b's n-rectangle with the following property:

$$Score(t_b,q) = \max_{t \in T_b} Score(q, t)$$

where T_b is the set of all potential tuples in the n-rectangle associated with bucket b. The S_q score that Restarts computes is the highest score that might result in no restarts in Step (5) of the algorithm above. In other words, using a value for Sq that is higher than that of the Restarts strategy will always result in restarts.In practice the Restarts strategy results in virtually all cases, hence its name.

Lemma 2: Let q be a top-k query over a relation R. Let S_q be the search score computed by strategy Restarts for q. Then, there are fewer than k tuples t in R such that Score(q, t) > Sq.

4.2 Choice of Selection Query C_q: Once we have determined the search score S_q, the algorithm in Section 4 uses a query C_q to retrieve all tuples t such that Score(q, t) ≥ Sq, where q is the original top-k query, and Score is the scoring function being used. In this section we describe how to define query C_q.

Ideally, we would like to ask our database system to return exactly those tuples t such that Score(q, t) ≥Sq. Unfortunately, indexing structures in relational database management systems do not natively support this kind of predicates. Our approach is to build C_q as a simple selection condition defining an n-rectangle. In other words, we define C_q as a query of the form:

```
SELECT * FROM R
WHERE (a1<=A1<=b1) AND ... ..AND (an<=An<=bn)
```

The n-rectangle [a1, b1] x ……x [an, bn] in C_q should tightly enclose all tuples t in R with Score(q; t) ≥ Sq. Given a search score S_q, the n-rectangle [a1, b1]x…..x[an,bn] that determines C_q follows directly from the scoring function used, the search score S_q, and the query q.

4.3 An Alternative Mapping Strategy:

This section adapts Fagin's A' algorithm to produce a new technique for mapping a top-k query into a traditional relational query. Unlike the Section 4.2 strategies, the selection query resulting from this new mapping is a disjunction, not a conjunction. Our goal is, again, to build a "one-shot" relational query that avoids restarts whenever possible. We proceed as in strategy NoRestarts to build a databas" with one tuple representing each bucket in the available n-dimensional histogram. We find the top tuples as in the NoRestarts strategy. We then compute an n-rectangle F = [a1,b1] x……x [an, bn] that encloses these top tuples tightly, and that has been extended so that it is "symmetric" with respect to the given query q. The tuples matching range [ai,bi] are the top tuples for q along attribute Ai.The selection query consists of the disjunction of the ai≤ Ai ≤ bi conditions. By retrieving all tuples that match at least one of these conditions, we retrieve the top tuples for each of the individual attributes. Furthermore, from the way we constructed F, there will be at least k tuples matching all n conditions. As with the original A' algorithm, we compute the score for all the one-dimensional matches. The k retrieved tuples having the highest score for q are the final answer to the original top-k query. The correctness of this algorithm follows from that of algorithm A' [5].

V. Top-k Queries with Set-Defined Selections

Given a relation R over the attributes {id,A1,…..,AN}, in which id is the unique identifier of an item (e.g., real world entity, document, video, image). The attributes Ai describe properties of an item and are numeric.

Definition: A top-K selection query is defined by the triple (K,Ai, S), that is, the size of the result ranking K,the attribute used for ranking Ai, and a set S ⊑ dom(id). The task is to efficiently compute those ids that have the K largest values for attribute Ai among all ids that appear in R and the query specific set S. The result is ordered by attribute Ai.

That means, a top-K query with set-defined selection can be expressed as a traditional top-K query over the subset of relation R that is given by the selection $\sigma_{id \in S}R$.

A SQL-like notation for this kind of query would look like

```
SELECT id, Ai FROM R
WHERE id IN S
```

ORDER BY Ai LIMIT K
where Ai is any of the numerical attributes of R. In this work, the case of a query specifying exactly one numerical attribute is considered.

5.1 Indices and cost Models: Considering a relation R with attributes {id,A1,…..,AN}, for each pair of attributes (id, Ai) two basic indices can be created:
- an index on the id attribute (called id-ordered index)
- an index on the numerical attribute Ai (called scoreordered index)

The id attribute are assumed to be densely populated in sequential order, such that the position of a score on the disk can be calculated directly from the id value. Hence, only scores need to be stored not the ids themselves. If the ids are not sequential, existing techniques based on B+ trees an be used for indexing. We adapt a column store data layout where the relation R is stored on disk in a per-attribute fashion (not row-by-row).

Both index organizations come with advantages and disadvantages: The id-ordered index is ideal if the size of the query set is rather small, resulting in a small number of index lookups. In contrast, the scored-ordered index is ideal if the size of the query set is large, such that K items out of the query set are found very early when scanning the sorted list on disk. To benefit from both sweet spots at the same time, a cost model is required to decide at query time which index to use. We optimize for low query response time, which is modelled as

$$t = c1 + c2 \times Db$$

where Db is the size of the data read from disk and c1 and c2 are constants which minimize the squared error on realworld measurements. The intuition is that c2 represents the data transfer time, while c1 approximates the time needed for a random access to disk. To keep the analysis tractable, we ignore inuences of distributions of ids and scores, and, hence, treat the size of the selection set and the value of K (specified in the query) as the main ingredients for the amount of data read from disk. The latter is represented in number of disk blocks, as the access to disk is naturally done in a block-based manner.

Once the query is submitted to the system, the execution time estimates are calculated and the index with the smaller execution time is used to answer the query. This procedure is totally hidden from the application layer: it appears as one index that combines the best of two index organizations. We call this index combined index.

5.2 Partitioned Index Organization: The above model provides a solid mechanism to identify the best index to use. The main idea of a partitioned index is to organize the original index into multiple chunks, such that a large fraction of queries is answered by reading only from one of them. This has a high potential: partitioning the score-ordered index into m parts lowers the query answering time by a factor of m (the number of blocks read from the disk would be m times smaller). The large score-ordered index is chopped up in a set of non-overlapping partitions. Each partition is organized as a score-ordered index. The decision which tuples to put together in a partition is done using a graph-based clustering approach. To fully harness such a partitioning, we check at query time whether the selection set is
i) entirely contained in one partition,
ii) mostly contained in one partition,
iii) distributed between many partitions.

Answering a query in the first case is done using only the selected partition, while the second case requires a lookup of missing tuples using the id-ordered index. In the third case
the query is answered using the combined index.To determine if the partitioned index should be used for query answering, we employ the cost models introduced in Section 5.1. If one partition captures the entire selection set, only the model for the partitioned index is used, which essentially is the model for a score-ordered index, where the index size is adjusted accordingly. In case not all of the ids from the selection set are found in one partition, the intersection size is used to estimate the query response time of the partitioned index. The number of the remaining ids is used to estimate the lookup cost of the scores in the idordered
index. The sum of the two estimates is used as a final response time estimate in case of this mixed access to the partitioned and the id-ordered index.

5.2.1 The Partition Selection Phase: To determine the partition which contains the largest subset of the selection set, data structures for computing set intersections are required. A bit-set structure is created for each partition, where each id is represented by one bit. The bit indicates whether or not the id is contained in the partition.This representation is exact (no false positives, no false negatives).
Bits Sets in Main Memory: Bit sets in main memory allow a fast calculation of the intersection between the query selection set and each of the partitions in the index. Only the partition with the largest intersection is used, in case the time estimate using this partition is less than the time estimate for the combined index. Otherwise the

query is answered using the combined index. This cost assessment is easy to achieve as the available bit sets give precise (exact) numbers of the contained and missing ids in a partition.
Bits Sets on Disk: If the bit sets do not fit in main memory, reading them from disk would in most cases consume more time than answering a query using the combined index. Our solution to this problem keeps only compact sketches [21,22,20] of partitions in main memory and the full sketch information on disk in the header of the corresponding partition. At query time, these sketches are used to determine the most promising partition to access by estimating the intersection size between partitions and the query selection sets.

Having only rough sketches of the partition contents in main memory requires changes to the querying algorithm. First, the most promising partition is identified using sketches. Then, the bit set for this partition is read and used to calculate the exact intersection size between the partition and the selection set. This information is then used to estimate if the selected partition is beneficial for answering the query. It is important to avoid accessing a partition on disk that turns out to be of little use once the bit set is inspected. To limit these wrong decisions, made by the estimation inaccuracy of the sketches, a partition is identified as promising only if we are highly confident that it will be useful for the query optimization later on.

5.2.2 Index Partitioning: The problem of data partitioning is formulated as follows: given selection sets from a query log, create m disjoint data partitions such that the probability of finding a randomly selected pair of ids from a randomly selected selection set in a single partition is maximized. The partitions should further be approximately equal in size. Determining the optimal number of partitions is not trivial: A large number of ideal partitions (i.e., each selection set is completely found in one partition) would decrease the runtime. However, increasing the number of partitions would increase the error introduced by the partitioning, that means, less and less queries could be answered by a single partition. For partitioning, we employ a technique used in recent work on data partitioning in distributed database systems, by Curino et al. [19]. The basic idea of their approach, coined Schism, is to create a graph based on a database workload (query logs). The vertices of the graph are tuples with edges connecting frequently co-occurring tuples in the transactions. The edge weight is given by the number of transactions in which two connected tuples occur together. Once the graph is constructed, the actual partitioning is done using constrained k-way graph partitioning. The basic assumption behind index partitioning is that selection sets are clustered in a meaningful way. Although this might not hold in general, selection sets are usually coherent in a semantic way. Constrained k-way graph partitioning is NP-complete, but there exist efficient and accurate approximation techniques.

VI. Approximate Query Answering

Given the partitioned index organization, even higher performance gains can be achieved by returning approximate top-k results instead of the exact ones. By approximate top-k results, we refer to the case when the selected partition does not cover all of the ids from the selection set. The missing ones could be retrieved based on the id-ordered index, but this is not done now. Hence, there is a risk that some of the missing ids would contribute to the actual top-k result, in which case the returned result is not exact. Although such approximations bring performance gains, without a quantification of the expected error, such approximate results are in most cases not acceptable, as the result quality can arbitrarily vary.

Approximate results are often very acceptable, but only up to a point where the precision is still above a certain level, for instance, above 80%. With precision we refer to the fraction of the returned top-k results which are also in the hypothetically exact result.

VII. Conclusions

In this paper, we studied the problem of mapping a top-k query on a relational database to a traditional selection query such that the mapping is “tight," i.e., we retrieve as few tuples as possible. Our mapping algorithms exploit the histogram structures and are able to cope with a wide variety of scoring functions. Our focus in this paper has been primarily on queries over continuous attributes.

In this work, we also addressed the problem of finding the top-k items out of a global index, where the choice of result items is restricted to a query-dependent subset. This problem is very fundamental: it appears in cases of distributed services on the Web or at the source layer of rank-aware query processing in databases, etc. To our knowledge, we are the first to consider this problem, which is completely different from common top-k aggregation queries in the literature. Our approach is based on a careful analysis of the pros and cons of an id- vs. a scoreordered index. We derived a cost model to choose the most suitable of these indices at runtime. We describe a way to benefit from a partitioned data organization and an approximate query answering. The conducted performance evaluation, based on real-world as well as synthetically generated data, revealed that the cost model is (almost) perfect in deciding which index to choose.

References

[1] G. Salton and M. J. McGill. Introduction to modern information retrieval. McGraw-Hill, 1983.

[2] M. J. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In Proceedings of the Twenty-fourth International Conference on Very Large Databases (VLDB'98), Aug. 1998.

[3] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96), June 1996.

[4] M. J. Carey and D. Kossmann. On saying \Enough Already!" in SQL. In Proceedings of the 1997 ACM International Conference on Management of Data (SIGMOD'97), May 1997.

[5] R. Fagin. Combining fuzzy information from multiple systems. In Proceedings of the Fifteenth ACM Symposium on Principles of Database Systems (PODS'96), June 1996.

[6] R. Fagin. Fuzzy queries in multimedia database systems. In Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems (PODS'98), June 1998.

[7] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In Proceedings of the Twenty-second International Conference on Very Large Databases (VLDB'96), Sept. 1996.

[8] L. Gravano and H. Garc__a-Molina. Merging ranks from heterogeneous Internet sources. In Proceedings of the Twenty-third International Conference on Very Large Databases (VLDB'97), Aug. 1997.

[9] W. Meng, K.-L. Liu, C. Yu, X.Wang, Y. Chang, and N. Rishe. Determining text databases to search in the Internet. In Proceedings of the Twenty-fourth International Conference on Very Large Databases (VLDB'98), Aug. 1998.

[10] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. In SIGMOD Conference, 2005.

[11] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv., 2008.

[12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci., 2003.

[13] U. G□untzer, W.-T. Balke, and W. Kie_ling. Optimizing multi-feature queries for image databases. In VLDB, 2000.

[14] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. VLDB J., 2004.

[15] M. Wu, L. Berti-Equille, A. Marian, C. M. Procopiuc, and D. Srivastava. Processing top-k join queries. PVLDB, 2010.

[16] V. R. Narasayya, H. Park, and M. Syamala. Automatic workload driven index defragmentation. PVLDB, 2011.

[17] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In CIDR, 2007.

[18] X. Wang, T. Malik, R. C. Burns, S. Papadomanolakis, and A. Ailamaki. A workload-driven unit of cache replacement for mid-tier database caching. In DASFAA, 2007.

[19] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. PVLDB, 2010.

[20] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci., 1985.

[21] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In STOC, 1998.

[22] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In RANDOM, 2002.