Aggregate Predicate Support in DBMS

Apostol (Paul) Natsev[*†]
IBM Watson Research Center
Hawthorne, NY 10606
Email: natsev@us.ibm.com

Gene Y. C. Fuh
IBM DBTI/390
San Jose, CA 95141
Email: fuh@us.ibm.com

Weidong Chen[†]
LinkAir Communications, Inc.
San Jose, CA 95032
Email: weidong@linkair.com

Chi-Huang Chiu[†]
National Chiao-Tung Univ.
Hsin-Tsu, Taiwan
Email: chchiu@cis.nctu.edu.tw

Jeffrey S. Vitter
Duke University, Box 90129
Durham, NC 27708
Email: jsv@cs.duke.edu

Abstract

In this paper we consider aggregate predicates and their support in database systems. Aggregate predicates are the predicate equivalent to aggregate functions in that they can be used to search for tuples that satisfy some aggregate property over a set of tuples (as opposed to simply computing an aggregate property over a set of tuples). The importance of aggregate predicates is exemplified by many modern applications that require ranked search, or top-k queries. Such queries are the norm in multimedia and spatial databases.

In order to support the concept of aggregate predicates in DBMS, we introduce several extensions in the query language and the database engine. Specifically, we extend the SQL syntax to handle aggregate predicates and work out the semantics of such extensions so that they behave correctly in the existing database model. We also propose a a new rk_SORT operator into the database engine, and study relevant indexing and query optimization issues.

Our approach provides several advantages, including enhanced usability and improved performance. By supporting aggregate predicates natively in the database engine, we are able to reuse existing indexing and query optimization techniques, without sacrificing generality or incurring the runtime overhead of database-external approaches. To the best of our knowledge, the proposed framework is the first to support user-defined indexing with aggregate predicates and search based upon user-defined ranking. We also provide empirical results from a simulation study that validates the effectiveness of our approach.

Keywords: aggregate predicates, nearest neighbor, query optimization

1 Introduction

In traditional database systems, users can limit the results of queries by using the standard relational and logical operators ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$, AND, OR, NOT). In addition, object relational databases, such as IBM's DB2, allow users to define their own predicates that can be used in queries and be exploited by query optimizers (Chen, Chow, Fuh, Grandbois, Jou, Mattos, Tran & Wang 1999). These predicates are scalar predicates and are *True* or *False* for individual tuples independent of other tuples.

Existing database systems support both scalar functions (e.g., ABS, SQRT) as well as aggregate functions (e.g., MAX, MIN, AVG). The main difference between the two types is that aggregate functions work over a set of tuples, while scalar functions take individual tuples as arguments. Unlike functions, however, existing database systems support only scalar predicates but not aggregate predicates. At the same time, many real world applications, especially in the multimedia domain, require aggregate predicates. The most well known example is ranked search applications, illustrated by the following types of queries:

- Find the top 10 images similar to a query image.
- Find the top 5 fault lines nearest to my house?
- For each store, find the top 10 selling products.

These examples share some common aspects. First, each query involves an *aggregate predicate*. In particular, we cannot determine if an image is in the answer set of the first query without comparing it with respect to other images in the database. Second, all examples involve a *search* based upon an aggregate predicate. In other words, we are not checking to see if a given image is the most similar one to the query image—instead, we are searching for the image most similar to the query image. Such queries are prevalent in multimedia and spatial applications, and are the focus of this work.

The rest of the paper is organized as follows. In the remaining part of this section we discuss related work on the ranked search problem, and list some advantages and disadvantages of the alternative methods. We also outline the main contributions of our approach and compare it with the previous approaches. Section 2 defines the notion of aggregate predicates and introduces certain extensions to the SQL query language to handle aggregate predicates. We give several query examples and explain the syntax and semantics of aggregate predicates in SELECT statements, and clarify their relationship in the context of WHERE, HAVING, ORDER BY, and GROUP BY clauses. In Section 3 we describe the runtime model and introduce a new rk_SORT operator. Section 4 deals with query optimization. In particular, we show how to extend the traditional query optimization techniques to the case of ranked search. Section 5 describes the experimental study we performed and lists empirical results to validate the effectiveness of our approach. We draw conclusions in Section 6.

1.1 Related work

The problem of nearest neighbor queries is well-known and has been studied extensively in the literature. One of the main approaches is to develop indexing mechanisms that target specific ap-

*CONTACT AUTHOR.

†The work was done while the author was at the Database Technology Institute, IBM Silicon Valley Laboratory.

plications of that problem. For example, evaluation of nearest neighbor queries has been investigated using the familiar R-trees in (Roussopoulos, Kelley & Vincent 1995). New indexing data structured have also been developed, such as SS-trees (White & Jain 1996). The Hybrid tree, although developed for a different purpose—namely indexing of high-dimensional data—can also support nearest neighbor queries, although the authors do not address the problem directly (Chakrabarti & Mehrotra 1998). While the approach of developing domain-specific indexing mechanisms works for specific applications, it has only been investigated for a very limited number of distance metrics, and it requires implementations of new tree facilities for search, concurrency, and recovery, for each new application. The cost of implementing such trees is very high, especially if they are to be used only for specific applications.

Another approach that alleviates the above problem is to develop generalized search trees that support nearest neighbor queries and ranked search. This approach is exemplified by the extension of GiST (Aoki 1997, Hellerstein, Naughton & Pfeiffer 1995), and it has the advantage of generality by allowing users to control every step of the traversal in the search tree. The disadvantage is that such user control is typically accomplished through expensive user-defined functions. The invocation of user-defined functions at each step of the tree traversal can cause heavy runtime overhead.

Due to the high cost of implementing domain-specific approaches or new indexing techniques, many applications use a simple application-driven approach, where the rank is explicitly computed for each tuple in the database, all tuples are then sorted, and only the top k are fetched by the application. This naive approach has high computation overhead if the number of desired responses is very small compared to the total number of records in the table.

A fourth approach is to reduce the top-k selection queries to a normal range query by estimating a cut-off threshold for the score used for ranking. Ideally, a top-k selection query based on a scoring function over a set of attributes A is transformed into an equivalent query with predicate $Score(A) > \tau$, where τ is the optimal cut-off threshold (i.e., the score of the kth best answer). In that case, there is no longer a need to sort all tuples in order to answer the query. Further savings can be achieved if there is also an index over the attributes A that can be exploited to fetch only those tuples that satisfy the score range predicate. In practice, however, the optimal τ is hard to find without sorting all tuples so the goal is to approximate the optimal threshold at a relatively small cost. The approach of (Chaudhuri & Gravano 1999) uses heuristics and histogram statistics about the data distribution in order to estimate τ, while (Donjerkovic & Ramakrishnan 1999) uses a probabilistic model and chooses τ so that the expected cost of the query execution is minimized. Both methods, however, suffer in the case of poor guesses since they lead to ranges that are either too small (with insufficient number of responses to the query) or too large (with more tuples than necessary).

If the estimated range does not produce enough answers to the query, the whole process is restarted and much of the work is done again, without being able to reuse the answers returned in the previous attempt. The reason for this computation overhead is the fact that the whole process is performed outside of the database engine and repeated range estimates lead to completely independent queries. The problem of incorporating these approaches into the database engine in order to eliminate this computation overhead turns out to be a non-trivial task however. First, it is not clear what the semantics of these approaches would be in relational database systems that are based on non-fuzzy set algebra and are traditionally built in the spirit of a push data model. Incorporating fuzzy functionality and ordered sets inside the database engine poses some semantic and optimization issues with respect to the proper interpretation and execution of such fuzzy operators. In addition, lack of accurate statistic information about the data distribution is a major obstacle for the above approaches since it leads to large errors when approximating τ. In contrast, the approach we propose in this paper is well-defined semantically and syntactically, works entirely within the database engine, and provides optimizations even in the case of inaccurate statistic information.

Carey and Kossman (Kossmann & Carey 1997, Carey & Kossmann 1998) were the first to push ranked sort into the database engine and to introduce early termination through a new STOP AFTER operator. They showed that considerable savings can be achieved by pushing that operator down the access plan tree, when the number of matches, k, is significantly smaller than the size of the relation. They provided two heuristics for the placement of the STOP AFTER operator—a conservative one that avoids restarts by placing the operator higher in the access plan tree, and an aggressive one that pushes the operator more deeply into the tree at the risk of having to re-execute the query in some cases. The disadvantage of the conservative approach is that it does not fully realize the savings that can be achieved from early termination, while the aggressive approach gets penalized when the cardinality is overestimated at an intermediate stage of the execution plan and the query needs to be restarted. The authors later extended their approach by using range partitioning techniques which allowed only partial re-computations (Carey & Kossmann 1998). The basic idea is similar to the approaches in (Chaudhuri & Gravano 1999, Donjerkovic & Ramakrishnan 1999) and it involves estimating range cardinalities so that the data can be partitioned into independent buckets, and all computation (including restarts) is done only on the relevant buckets, thus avoiding overhead for irrelevant buckets. This approach alleviates the problem of insufficient distribution statistics somewhat because even if the partitioning is not optimal, the system can still prune some irrelevant buckets. However, the amount of data that gets pruned is still highly dependent on the partitioning and therefore on the distribution statistics. Such statistics are generally unavailable whenever the ranking is done according to a user-defined function for example. In addition, for partitioning purposes, the approach requires the score value for each tuple, and therefore, needs to do a full table scan and evaluate the user-defined predicate on each tuple. The major cost savings come from the fact that sorting is avoided for some of the tuples but the data still needs to be scanned completely. Ideally, we would like to avoid that if there is an index available for the predicate used for ranking.

In order to exploit the full benefits of early termination in the case of user-defined indexing and ranking, the system simply must delegate some decisions to the user. This paper proposes an approach similar to the above one but with several distinctions. First, it allows index exploitation so that savings can be achieved not only from avoiding to sort unnecessary data but also from avoiding to scan such data. Most importantly, such index exploitation is available even for user-defined indexes and when the sorting is based on a user-defined predicate. This opens up more opportunities for query optimization based on early termination, and is the main advantage of our approach

over that of Carey and Kossman—by utilizing the existing framework for user-extensible indexing, we are able to exploit such indexes and achieve early termination even if no runtime system statistics are available. In addition, we extend the functionality of the ranked search so that the sorting is performed over an independent aggregation window, thus enabling a richer set of queries.

1.2 Proposed approach and advantages

This paper makes several contributions towards aggregate predicates and ranked search in DBMS:

- We extend the SQL language with the concept of ranked search and aggregate predicates. We work out the syntax and semantics of aggregate predicates so that they can be used in the same way as traditional scalar predicates without disturbing the meaning of WHERE, HAVING, GROUP BY, and ORDER BY clauses.
- We design a flexible language mechanism (similar to OLAP window specification) that allows aggregate predicates to operate over an arbitrary set of tuples
- We design runtime query execution strategies for aggregate predicates that support both ranked incremental retrieval with no explicit bound and ranked incremental retrieval with a specific bound, such as top 10.
- We design indexing and query optimization strategies for aggregate predicates when the rank in an aggregate predicate is computed using only system built-in functions.
- We extend the above scenario to the previously unsupported case of user-defined functions, and we support extensible indexing with aggregate predicates.

The main advantage of our proposal is the native support of aggregate predicates with better performance, as compared to some alternative solutions. The introduction of aggregate predicates into the SQL language provides enhanced expressive power for users. While users may be able to simulate some aggregate predicates using aggregate functions and relational operators, such an approach has a usability problem and a performance problem. The reason is that aggregate functions and aggregate predicates serve different purposes. For usability considerations, if a user wants to find the top 5 images most similar to a query image, the user would have to create an explicit column for image similarity ranking, and then create a condition on the explicit rank column. This leads to significant rank maintenance problems with respect to data insertions and deletions—in particular, the user would have to recompute the rank column every time data is updated. In terms of performance, aggregate functions are always computed and involve scanning of the entire table, while search based on aggregate predicates can avoid or limit such computations and data scan.

We believe that the native support of ranked search is a significant advantage in practice. For example, consider an improvement to the naive approach, where the application controls the number of records being processed for ranking by doing the computation in rounds. This is essentially the same as the approach we are proposing but implemented entirely through a stored procedure or a client command. The disadvantages of using this approach outside of the database engine are several. In terms of usability, not all of the functionality can be achieved that way. For example, nested queries involving aggregate predicates would not be possible because the stored procedure or client command cannot put the ranked output into the pipeline. In addition, Boolean expressions involving aggregate predicates would either be unavailable or would have to be implemented as separate stored procedures or client commands. This would result in a significantly more limited query power. On the other hand, pushing the functionality into the database engine enhances the query power and reuses existing indexing and optimization techniques for the case of aggregate predicates. In terms of performance, implementing the approach outside the engine would mean using separate SQL queries for each round. In addition to potential network delay, this would also lead to increased compilation and initialization time, and therefore degraded efficiency. The performance benefits of a database engine approach have been previously investigated in (Kossmann & Carey 1997) for example. Finally, the database-external approach requires a separate stored procedure/client command for each index type, which is programming labor consuming and entails high maintenance cost.

Compared to the alternative methods, our approach is most similar to the work of Carey and Kossman (Kossmann & Carey 1997, Carey & Kossmann 1998) in that it uses early termination and pushes the ranked sort computation down the plan tree. However, in our approach this is achieved through the introduction of a new and more powerful rk_SORT operator, which enables query optimization and exploitation of user-defined indexes. This helps us to reduce not only sorting costs but also data scanning costs. In contrast to other related work, our approach is extensible in the sense that users can control ranked search, thus avoiding some of the shortcomings of domain-specific approaches. Such control is furthermore exercised at a high level in our framework by generating a sequence of search ranges that are used by the underlying access method without compromising its efficiency. This avoids the runtime overhead of the GiST approach, for example. Also, our approach is dynamic and incremental—it handles data insertions and deletions that make the rank hard to maintain by some of the other approaches. Finally, it does not rely heavily on distribution statistics, although it can make use of such statistics, if available. Using data distribution information and heuristics, the rk_SORT operator could minimize the number of rounds, thus avoiding some initialization overhead and restricting unnecessary work to a minimum.

2 Language model

In this section we describe the SQL extensions necessary to support aggregate predicates. Informally, we define aggregate predicates as functions $AP(W, \ldots) \rightarrow \{True, False\}$, where W denotes a window specification (i.e., the first argument is a set argument), and the dots denote one or more scalar arguments. In other words, aggregate predicates operate over a set of items and compute a true or a false value based on the scalar attributes with respect to the specified window. Usually, this means searching for some tuples that satisfy a certain aggregate property with respect to the search window, such as the top 10 salaries in a department. Note that aggregate predicates are more general than ranked search but for simplicity we refer to aggregate predicates only as ranked search in this paper and use the two interchangeably.

Our approach of supporting aggregate predicates is to treat them the same way as scalar predicates (e.g., relational or logical operators). Consider a scalar predicate such as $c_1 < c_2 + 10$. To determine

the truth value of this predicate, all we need are the values for c_1 and c_2. In contrast, an aggregate predicate needs more information in order to determine its truth value. For instance, in order to find the top 5 salaries, we need the following information:

- the ranking used in the aggregate predicate: the attribute "salary" in this case
- the order we are interested in: descending for the top 5 or ascending for the bottom 5 salaries
- the number of answers we want: 5 in this case
- the set of values over which the aggregate predicate is evaluated (e.g., all salaries or per department)

We need to capture all of the required information in our extension to the SQL language. For this particular example, suppose that we have a table called *EMP*, with columns of *name*, *dept*, *salary*, *age*. The following query requests the top five salaries and the corresponding names:

```
SELECT name, salary
FROM   EMP
WHERE RANK(salary) DESC FIRST 5
```

If we need to retrieve the top five salaries in each department, we can use the following syntax, similarly to the way window specification is done with OLAP functions:

```
SELECT dept, name, salary
FROM   EMP
WHERE RANK(salary) DESC FIRST 5
      OVER(dept)
```

There is a question of how aggregate predicates interact with scalar predicates. Consider the following query:

```
SELECT dept, name, salary
FROM   EMP
WHERE RANK(salary) DESC FIRST 5
      OVER(dept)
      AND age < 25
```

What is the semantics of this query? Since the aggregate and the scalar predicate are on the same level, the semantics is to find the department, name, and salary of employees in each department who earn one of the top five salaries in the department and who are younger than 25. What if we want the top five salaries among young employees instead? The following query can be used for this purpose:

```
SELECT dept, name, salary
FROM   EMP
WHERE  age < 25
HAVING RANK(salary) DESC FIRST 5
       OVER(dept)
```

From the above examples, it may seem that aggregate predicates have different semantics in WHERE clauses and in HAVING clauses. This is not the case! The aggregate predicates, just like any other predicate, have unique semantic meanings and can appear in both types of clauses. It is the different semantics of the WHERE/HAVING clauses that cause the query to be interpreted differently since they impose different order on the predicates. Note that when the HAVING clause does not have any aggregate predicates, all scalar predicates in the clause can usually be merged with the predicates in the WHERE clause. Such merge, however, cannot be allowed if the HAVING clause has at least one aggregate predicate.

The syntax informally introduced above is well defined even if the statement contains GROUP BY or ORDER BY clauses. The RANK term will still interact nicely with such clauses because aggregate predicates have their own group specification (indicated by the OVER term). A potential problem for the rank search syntax arises when we have more than k tuples that satisfy the aggregate predicate. For the salary example, what if we have the top fifth and sixth salary being actually the same? Do we include the sixth in the answer set as well, or do we choose one of them randomly? One possible solution, used with OLAP functions, for example, is to introduce another keyword, (e.g., RANKGAP), to indicate the desired behavior.

So far we have considered only isolated aggregate predicates. Just like scalar predicates, however, aggregate predicates can be combined with others using logical operators, such as AND and OR. Note that such combinations of aggregate predicates are different from those considered by Fagin (Fagin 1998). To support compound aggregate predicates like those in (Fagin 1998), we can use the following informal syntax:

```
SELECT dept, name, salary
FROM   EMP
WHERE RANK(salary) DESC AND
      RANK (age) ASC FIRST 5
      OVER(dept)
```

In this query we are trying to find, for each department, the top five young and high salaried employees. The ranking of a compound aggregate predicate depends on how the ranking of each component aggregate predicate is combined with each other. In such a case, we may want to decide how much weight to give to each component aggregate predicate, and introduce some new syntax for that. For our purposes, however, we are not interested in such compound aggregate predicates, and for Boolean combinations of aggregate predicates we adopt the semantics dictated by set theory. For simplicity, in the rest of the paper, we do not consider Boolean combinations of multiple aggregate predicates.

Formally, we introduce a new syntax, **aggregate predicate**, into the "search condition" of the SQL language. The extended grammar is as follows:

```
predicate :=              scalar-predicate |
                          aggregate-predicate
aggregate-predicate :=    RANK (scalar-expression)
                          [ASC | DESC]
                          [FIRST numeric-expr.]
                          [OVER (aggregate-window)]
```

The RANK word identifies the scalar expression as an aggregate predicate, the order of the ranking is done according to the ASC(ending) or DESC(ending) keyword (ASC by default), and the number of desired results is specified in the FIRST clause. If the FIRST clause is missing, all results are output in the specified order. The window specification in the OVER clause, if present, determines how the aggregation is to be performed. If no aggregation window is specified, the result is aggregated over the entire table.

Let *distance* be a user-define function (UDF) that computes the distance between two points. The following query finds the nearest 10 ATMs from a given location using an "aggregate predicate" syntax in the WHERE clause:

```
SELECT a.name, a.location, a.address
FROM   ATM a
WHERE RANK(distance(a.location, :whereIam))
      FIRST 10
```

Intuitively, the query engine fetches the tuples in the ATM table and sorts them based on the value of the UDF invocation in the WHERE clause. The top 10 tuples in the default sort order are returned as the result. If there are less than 10 records, the entire table is returned. Score ties among tuples for the 10th place are broken arbitrarily.

Just like normal predicates, aggregate predicates can appear not only in the WHERE clause but also in the HAVING clause of a query, as shown in the following example:

```
SELECT s.id, c.name, c.address, c.location
FROM   customers c, stores s
WHERE  within(c.location, s.zone) = 1
HAVING RANK(c.income)
         FIRST COUNT(*) * 0.1 OVER(s.id)
```

The result of the above query contains, for each store, the tuples that represent the top 10% of customers in terms of their annual income. The WHERE clause specifies the join condition between the two tables, while the HAVING clause specifies the aggregation to be done for each store.

3 Runtime model

In this section we describe the runtime model of our proposed framework and we discuss its implementation. In the current database model, the only available approach to dealing with ranked search is to **sort all** tuples in the database and then **fetch** k answers from the top. The main problem with this approach is that **all** tuples need to be processed and ranked, even if we need only a small subset of them. The reason is that the sorting and fetching phases are separate and the latter depends on the former. One natural solution is to combine the two phases in a sequence of independent rounds so that we can answer the query in the early rounds and avoid unnecessary computation. We can achieve this by introducing a new operator, rk_SORT, into the database engine. The rk_SORT operator is similar to the existing SORT operator but it is aware of the notion of rounds, and is capable of stopping or resuming computation at the end of a round. By pushing the rk_SORT operator deep into the database engine, we can also reuse all of the existing indexing and optimization techniques.

This additional functionality raises the question, however, of where the new operator fits in the current "push" model for databases. Traditionally, the query execution model has been one of pushing data from one end of the pipeline (the physical layout of the data) to the other (the output of the database query). The pipeline flow therefore stops only when there is no more data to be channeled through. In ranked search queries, though, we would like to terminate execution, as soon as we have enough answers, *even though there may be more data in the pipeline.* This leads us to reconsider the query execution model, and conceptually to think of a "pull" model, where data is pulled from the input towards the output end of the pipeline. This demand-driven model is already getting adopted in commercial systems in terms of functionality – all of the major DBMS support some form of top-k queries. Unfortunately, the architecture and implementation details for this kind of support in the commercial systems have not been published, and it is not clear if such support is provided to the extent of functionality only or with query optimization.

We now give a more formal description of our runtime model. We focus on support for aggregate predicates with extensible indexing because it is the harder case but exploitation of system built-in index types is also available. In that case, the role of the range producer below is performed by the query optimizer itself, who generates the sequence of ranges. User-defined indexing is described in the context of IBM's DB2 database system. In particular, we assume that the user is provided with certain hooks into the database engine in order to allow maintenance and exploitation of user-defined index types (Chen et al. 1999). The main such hook that is crucial to our framework is the range producing function. The range producer's task is to map a specific (possibly user-defined) predicate into a set of index key ranges that contain the tuples that satisfy the given predicate. Each user-defined predicate (such as *distance, within*, etc.) has an associated user-defined range producing function. The main premise of our approach is to isolate domain-specific knowledge into the range producer and to place the burden of searching onto that function.

As an example of how this might work, suppose that a database table contains two-dimensional data about ATM locations, stored in a grid index. The domain is therefore divided into grid blocks (perhaps organized in a multi-layer fashion) so that each grid block contains only the points that fall into that block. Given a query position, the nearest 5 ATMs will be located in the grid blocks nearest to the query point. It is therefore desirable to start processing of the grid blocks in a spiral-like order, unwinding from the query point outwards, until we find the nearest 5 ATMs. This suggests an intuitive order for generating search ranges. The only assumption we need in order to make this approach work is the requirement that we consider all points within a certain distance of the query point before we make a decision whether we have the top 5 answers or not. In other words, in the first iteration we generate a set of ranges that cover all points within a radius r_0 from the query point. We then sort only these points by their distance to the query point and output them. If we have more than 5 answers we terminate processing. Otherwise, we proceed to the next iteration, in which we increase the radius to $r_1 > r_0$ in order to include the next round of grid block ranges. Performing the processing in such ring-wise fashion limits the search space while guaranteeing the correctness of the answers. The selection of the radii for the successive iterations can be based on heuristic or probabilistic models, as in some of the approaches discussed in Section 1.1 (Chaudhuri & Gravano 1999, Donjerkovic & Ramakrishnan 1999). When there are no statistics available to help us choose the round radii, the system can revert to a simple heuristic of using the density from each round in order to guess next round's radius (e.g., see Section 5).

The above example refers to the spatial domain but can be generalized to arbitrary domains, including multimedia ones, with user-defined indexes and scoring measures used for ranking. The range producer generates a set of ranges R_i, $i \in [1, n]$, where $R_i = (begin_i, end_i, r_i)$ and $begin_i$ and end_i are the beginning and ending index keys, while r_i is a scalar measurement representing the round radius. Conceptually, the ranges are divided into "rounds" so that all ranges in a given round $\mathcal{A}$ have an identical radius, $r_{\mathcal{A}}$, which must be strictly bigger than the radius used in the previous round (we refer to this restriction as the "round monotonicity assumption"). Informally, the notion of a round refers to all tuples with scores less than or equal to the round's radius, as identified through the sequence of ranges. The range producer is responsible for making sure that all ranges with radii $\leq r_{\mathcal{A}}$ cover the tuples that fall into the round with radius $r_{\mathcal{A}}$. More formally, round $\mathcal{A}$ with radius $r_{\mathcal{A}}$ consists of all tuples (*key*, *score*, *data*) for which $score \leq r_{\mathcal{A}}$, where *key*, *score*, and *data* de-

rk_SORT (*str*(*Radius*), *k*, *groupByList*, *scoreFn*):

1. Initialize the round radius to -1 for all groups;
2. While not end-of-stream, do:
   (a) Fetch the next available tuple in *str*;
   (b) If the target group for the new tuple is frozen, discard the tuple and goto 2;
   (c) Let r be that tuple's *Radius* and let r_g be the *Radius* of that tuple's group;
   (d) If $r \neq r_g$ then
      i. If the number of tuples with $scoreFn \leq r_g$ is $\geq k$, then "freeze" the current group and goto step (a);
      ii. Set $r_g = r$;
   (e) Insert the new tuple into the sorted table based on *groupByList* and *scoreFn*, only if the tuple is not in the table already;
3. Pipe rank-sorted table to next stage in access plan

Figure 1: Pseudo code for the rk_SORT operator.

1. For each c_i in $index(c_1, c_2, \ldots, c_k)$, do:
   (a) $keyPredicates(i).ff = +\infty$;
   (b) $keyPredicateFound$ = False;
   (c) For each predicate $p_j \in P$, do:
      i. If p_j is not a key predicate or $target(p_j) \neq c_i$ continue;
      ii. $keyPredicateFound$ = True;
      iii. Let ff be the "filter factor" of p_j;
      iv. If $ff < keyPredicates(i).ff$ then
         A. $keyPredicates(i).ff = ff$;
         B. $keyPredicates(i) = p_j$;
2. Construct the ISCAN operator followed by the FETCH operator using the information in the *keyPredicates* array;
3. If an aggregate predicate is used in the index exploitation, then build the rk_SORT operator on top of the FETCH operator.

Figure 2: Pseudo code for index exploitation.

note the tuple's index key, measurement score used for ranking, and general data. The range producer ensures that $\exists i \in [1, n]$ s.t. $begin_i \leq key \leq end_i$ and $r_i \leq r_{\mathcal{A}}$. Given the above assumptions, the database engine can use the user-defined index to do the search efficiently, performing the range processing in the user-specified order and terminating as soon as k answers have been output.

The definition implies that all scores for tuples in a given round are smaller than or equal to that round's radius. The round monotonicity assumption above implies that $r_i < r_j$, $\forall i < j$. Both conditions together imply that if a tuple belongs to round i, it also belongs to all rounds j, for $j > i$. Thus, the definition of a round suggests an inclusive relationship between successive rounds. For performance purposes, however, it is more efficient to implement the range producer so that each successive round of ranges includes only ranges that have not been generated before. That way, at each iteration the range producer outputs only the successive round differences and generates disjoint ranges that collectively form the rounds. This restriction does not have to be enforced by the system so even if the range producer generates duplicate ranges, they can be removed automatically.

The runtime model described above is implemented through the introduction of a new rk_SORT operator. The rk_SORT operator is described by the pseudo code in Figure 1. It takes four inputs and produces as output a rank-sorted set of tuples. The first argument is the input stream *str*. A designated column for the *Radius* is associated with each tuple of the input stream. The second argument, k, determines the cardinality of the result set by specifying how many answers should be produced. The third argument, *groupByList*, is the list of expressions which define the formation of groups in the result set. The last argument, *scoreFn*, is the expression in the aggregate predicate that is used for ranking purposes (it appears right after the RANK keyword).

As described previously, the concept of round radius facilitates early termination of the rk_SORT operation in certain cases where the input stream is divided into multiple rounds. The idea is that if, at the end of round n, there have been at least k tuples with score less than or equal to the radius of round n, then the rk_SORT for that group can terminate by ignoring the remaining tuples in the group (i.e., freezing the group). Correctness is assured since all tuples in the following rounds are guaranteed to have scores worse than the current round radius. Taking 2-D search as an example, the search for candidates is intuitively performed based on a sequence of "rings" from inside out. The round radius is set to the distance between the point of interest and the outer circle of the ring. Therefore, change of round radius indicates change of the search ring. In the case where ring search is not possible or available, the *Radius* of the input stream should be set to infinity so that all tuples will be considered in the rk_SORT.

4 Query optimization

So far, we have described the runtime model for search based upon aggregate predicates, and we have introduced the necessary tools for database engine support of such aggregates. We now turn to the problem of query optimization in the new model and exploitation of indexes using aggregate predicates. For query optimization, we simply reuse the existing optimization techniques through index exploitation (Hass, Chang, Lohman, McPherson, Wilms, Lapis, Lindsay, Pirahesh, Carey & Shekia 1990, Sellinger, Astrahan, Chamberlin, Lorie & Price 1979, Stonebraker, Wong & Kreps 1976). A typical access plan for a JOIN query, for example, would join the corresponding tables, pipe the result to a SORT operator, group tuples if necessary, and output the result. The amount of savings achieved depends on how the JOIN is performed. For example, if there is an index defined on any of the join attributes (resp., the user-defined predicate used for joining the tables), then the corresponding table may become an inner JOIN table and the index could be exploited through a key predicate. For index exploitation purposes, the optimizer needs to identify a match between a predicate and an index, and then use both to generate ranges that limit the search space. In the case of aggregate predicates, that match means that the scoring expression in the predicate needs to be a user-defined predicate or an attribute with an associated index.

In the cases where no index is available (i.e., all table scan cases), the round radius passed on to the

rk_SORT operator is the infinity value and all records are fetched in a single round. The same holds for index exploitation cases where the key predicate used for index exploitation is not an aggregate predicate. In such cases, there is a partial index exploitation to filter out some records but there is no early termination based on ranking. We note that our framework does not force any access plans to the query optimizer but only provides more options to consider. Ultimately, the access plan used by the optimizer will be the one that has minimized expected cost of execution. In order to be able to fully exploit aggregate predicates for rank-based early termination, however, the following conditions must be met:

- There must be a match of the aggregate predicate to an index, either through a ranking attribute or UDF.
- The cardinality argument of the aggregate predicate must be known at execution time (e.g., it could be a numeric argument, a host variable, or a column from an outer table in a nested join).
- The access plan generated must have the rk_SORT operator immediately after the FETCH operator, following an index scan (ISCAN) with the aggregate predicate being used as a key predicate.

We now give a more formal overview of the modified index exploitation process for a JOIN query. Without loss of generality, we only consider left-deep join trees. Also, note that when the join composite is the empty set, everything is still well defined and we have the special case of a non-join query. The definitions below, as well as the basic idea of the index exploitation algorithm of Figure 2, are borrowed from the standard literature on query optimization (e.g., see (Hass et al. 1990, Sellinger et al. 1979, Stonebraker et al. 1976)). Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of tables to be joined. Let $C = \{t_{c_1}, t_{c_2}, \ldots, t_{c_m}\}$ be the set of tables in a join composite, where $m < n$ and t_{c_i} belongs to T and $t_{c_i} \neq t_{c_j}$, $\forall i \neq j$. Let t be the new table to be joined to the composite C. Let $P = \{p_1, p_2, \ldots, p_l\}$ be the set of predicates of the entire query block. For each $p_i \in P$, let $TabRef(p_i)$ be the set of tables referenced in p_i. Predicate p_i is *eligible* with respect to (C, t) *iff* $TabRef(p_i) \in C \cup \{t\}$. In this case, we denote the column being constrained as $target(p_i)$ and the constraint caused by p_i as $bound(p_i)$. For example, $target(salary > 50000)$ is the column "*salary*" and $bound(salary > 50000)$ is the literal 50000. Predicate p_i is a *join predicate* between C and t if p_i is eligible and $t \in TabRef(p_i)$ and $TabRef(p_i) \cap C \neq \emptyset$. Predicate p_i is a *local predicate* of t if $TabRef(p_i) = t$. Predicate p_i is a *key predicate* on $index(c_1, c_2, \ldots, c_k)$ of table t if $target(p_i)$ is a key column and $bound(p_i)$ is invariant given C as the join composite. In particular, an aggregate predicate ap: "RANK(exp_1) FIRST exp_2" is a key predicate if exp_1 is a key column or it is a user-defined predicate of which $target(ap)$ is a key column, and $bound(ap)$, as well as exp_2, are invariant with respect to C. The index exploitation algorithm is listed in Figure 2.

Figure 3 depicts two possible access plans available to the query optimizer for one of the most basic scenarios involving aggregate predicates. The query being optimized selects the nearest 10 customers to the IBM Silicon Valley Laboratory (i.e., the :*svl* location). An example of another query of the same type is given below:

```
SELECT *
FROM   customer C
WHERE RANK(C.income) DESC FIRST 10
```

This query selects the top 10 customers with highest income but it uses a table attribute rather than a UDF as the aggregate predicate. Both of these queries are similar because the only possible optimization of the predicate evaluation is early termination through index exploitation. This is so because neither query contains joins, aggregate functions, Boolean expressions, GROUP BY, ORDER BY, or HAVING clauses. However, the two queries differ in that the first one would have to exploit a user-defined index, while the second one would exploit a system index. Figure 3 shows the two access plans the query optimizer will have to choose from. If an index is available on the appropriate attribute ($C.loc$ or $C.income$, resp.), then the optimizer would generate the access plan with index exploitation. Due to space considerations, we do not discuss here access plan choices for more complicated queries. However, more details are available upon request from the authors.

5 Performance

In this section, we evaluate several ranked search application scenarios and compare our approach with alternative methods using simulation experiments. For the experiments in this section, we used IBM's DataJoiner v.2.1 running on an RS/6000 (133 MHz) Model E30 workstation with 576 MB of RAM. We used synthetic data for our experiments consisting of one million tuples in a customers table with attributes *id*, *location*, *name*, *salary*, *address*, *city*, *state*, *zip*, and *category*. The types of the attributes are straightforward, with the exception of the *location* type, which is a two-dimensional point. All customers' locations were distributed in a grid of size 10000×10000 in a non-uniform fashion with concentration around the origin. A spatial grid index was built on the location column using IBM's Spatial Extender. Due to space and time considerations, we do not report experiments using built-in index types but only with extended index types, such as the grid index extension for 2-dimensional data defined by the IBM Spatial Extender. We also do not report comparisons with the naive approach of sorting the entire dataset and fetching only the top k answers. The performance benefits over that approach were illustrated in (Kossmann & Carey 1997), for example, and were confirmed by our observations.

We compared our approach to the alternative repeated-guess approach, which tries to guess an appropriate range that would return the top-k answers. If the guess is wrong, and fewer than k answers are returned, the process is repeated with a refined guess. The heuristic we used was to consider the overall data density, d, and based on that to estimate the range, r, that would return k answers (e.g., $r = \sqrt{k/d}$). If the guess was wrong and $k' < k$ answers were returned, then for the next guess we used density $d' = k'/r^2$. We also limited the maximum range increment to avoid situations where very low density in a certain area causes excessively large range guess for the next iteration. Our integrated ranked search approach was simulated by modifying the user-defined range producing function for the spatial grid index type defined in the Spatial Extender. The range producer was modified to generate the 2-D search ranges in the spiral order discussed previously. The same guessing heuristic was used to generate the sequence of ranges as in the repeated-guess approach but in that case, previous computation was reused, and the whole search was performed as a single query.

The experiments that we performed fall in the client-server scenario, where some output of a ranked search query needs to be returned to the user via the

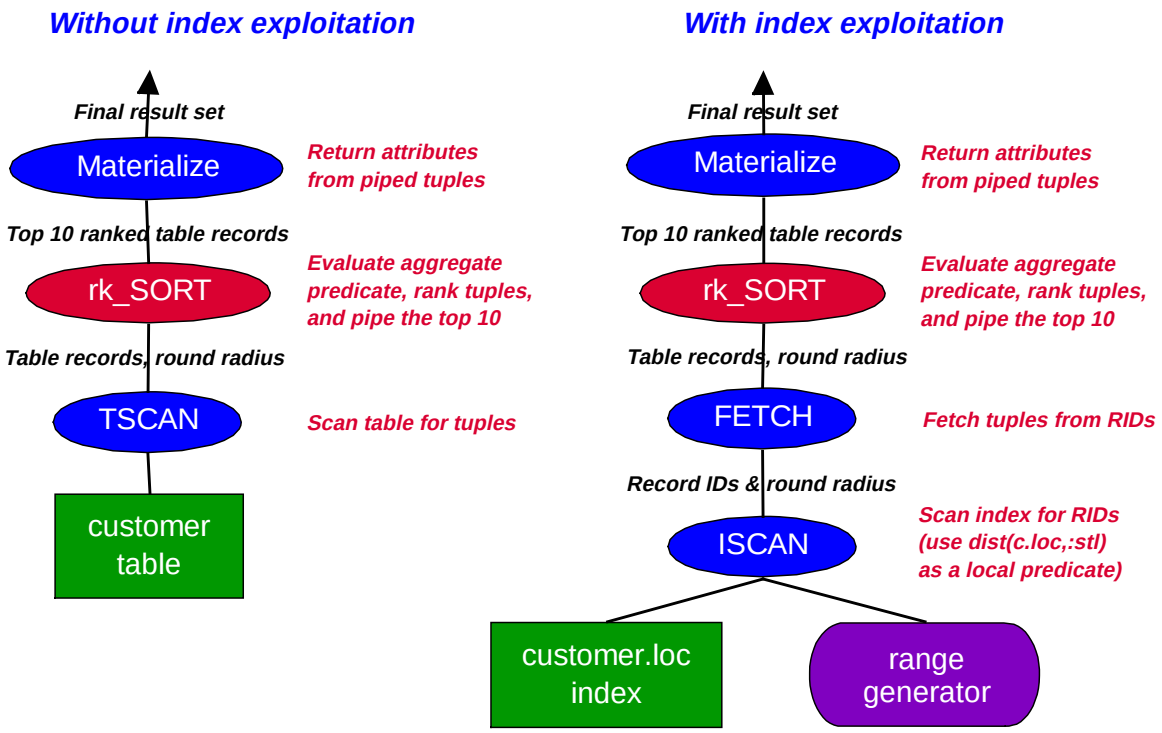
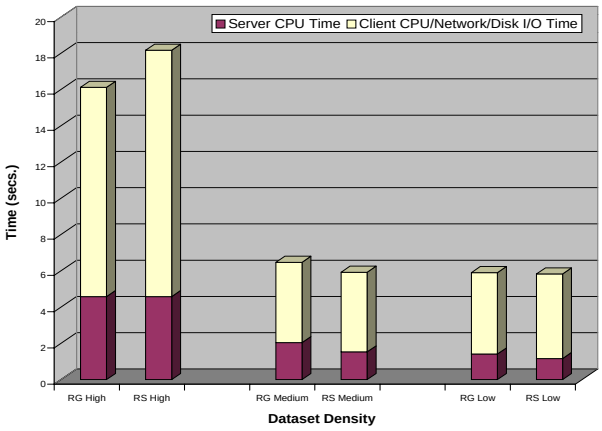


Figure 3: Simple aggregate predicate: with and without index exploitation. The SQL query is: **SELECT** $*$ **FROM** *customer* **WHERE RANK(dist(***customer.loc*, *:svl***)) FIRST 10**

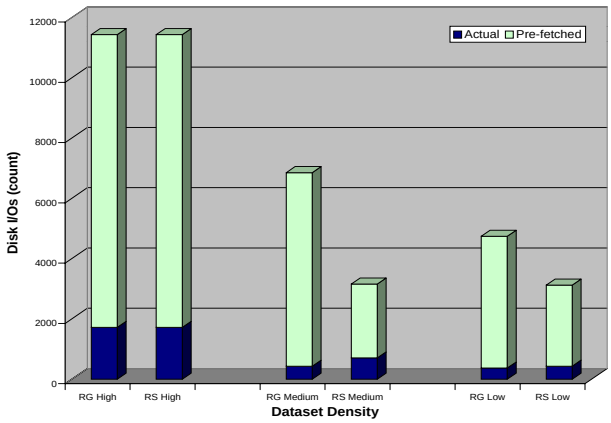
network. Since the CPU time spent on the client side was negligible, we only distinguish between server CPU time and everything else (including client side CPU time, network delay and disk I/O time). In addition, we provide data for the total number of logical page requests and the number of actual disk I/Os to illustrate the I/O behavior of each approach. We considered three different types of queries: top-100 (0.01% selectivity), top-1000 (0.1% selectivity), and top-10000 (1% selectivity) queries. The first type is representative of highly selective queries with very small filter factor that return all available information for just a few tuples. An example would be to return all the data for the top 100 employees with highest salaries. The second class of queries is what we call mid-range queries that request data from a moderately large number of tuples but require only some of the information available for each tuple. For example, we may be interested in compiling a mailing list consisting of the names and addresses of the closest 1000 customers to a given store. Finally, the third query type is representative of large-scale queries that request only aggregate information or certain statistics about a large subset of all tuples. An example in this case would be to get the average customer income of the closest 10 000 customers to a given prospective store location. Such information could be used, for example, to determine the best choice for a new store location. Queries with lower selectivities (i.e., higher filter factors) are typically used in a server-server scenario, where the returned data is inserted back into a table or a view for later referencing. In addition to a usability advantage of supporting such functionality with a single query, our approach would also have a performance gain in this scenario as well. However, since no database external approaches can support such nested queries, we do not report performance data for such queries.

The experimental results are presented in Figure 4. The left hand side shows timing results for the Repeated Guess (RG) and the integrated Ranked Sort (RS) approaches in three different data densities (high, medium, and low). Each bar illustrates the server-side CPU time (the lower portion of the bar), as well as the turn-around wall clock time (the entire bar). The right hand side of the figure shows the equivalent disk I/O comparisons for the same queries. The lower part of each bar depicts the number of actual I/Os for the corresponding approach, while the entire bar depicts the total number of disk page requests. The results are presented for each of the selectivity scenarios (top-100, top-1000, and top-10000 queries).

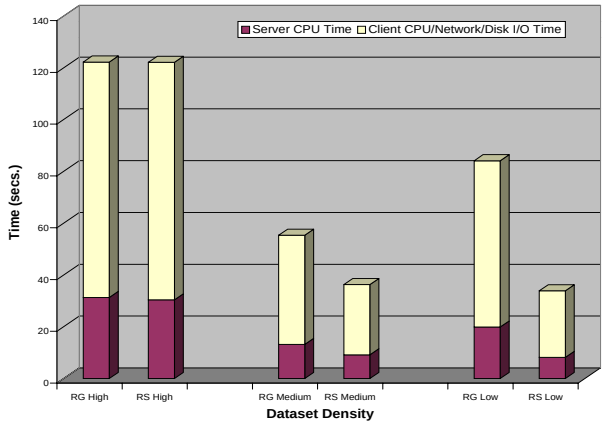
The results lead to several conclusions. The first is that the performance of both approaches is inversely proportional to the density of the dataset. This can be explained by the fact that the queries in dense areas underestimate the cardinality of the guessed ranges and therefore return more tuples than needed even in the very first round. In that case, the performance of both approaches is almost identical because the work done in both cases is the same. However, as the query area density decreases and approaches the overall dataset density, the range estimates become better and therefore both approaches exhibit improved performance, with the Ranked Sort approach outperforming the Repeated Guess approach due to less overhead for each round. Finally, at the low density scenario, both approaches keep overestimating the range cardinalities and thus the number of rounds (guesses) needed for fetching k answers increases. In the case of the Repeated Guess approach, this leads to performance degradation due to re-computation overhead, while the Ranked Sort approach retains its performance due to its low round overhead. This demonstrates that the proposed Ranked Sort approach is more robust with respect to query area density and less sensitive to range estimation errors. Also, the results show a consistent advantage of our approach over the alternative one, with the performance gap increasing for the queries with higher filter factors. The reason for this gap is the fact that queries with high filter factors need to fetch more data, and therefore, the wrong range guess leads to more rounds (i.e., recomputation overhead) in the case of the Repeated Guess approach.



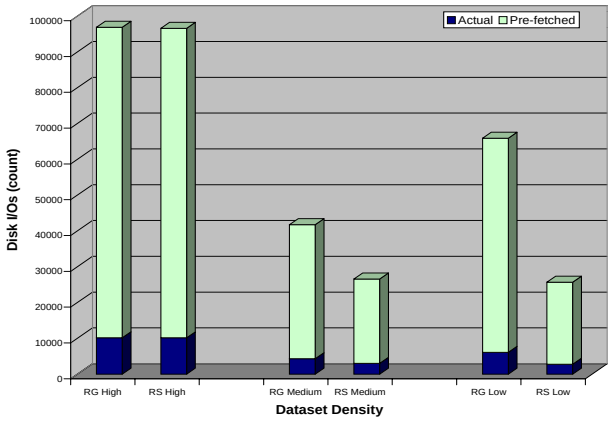
(a) Timings for top-100 queries (0.01% selectivity)



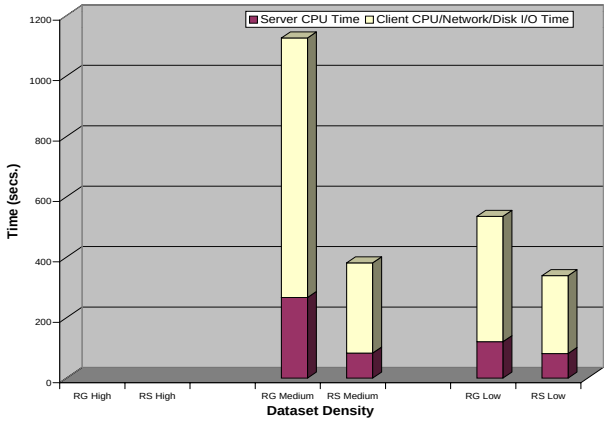
(b) Disk I/Os for top-100 queries (0.01% selectivity)



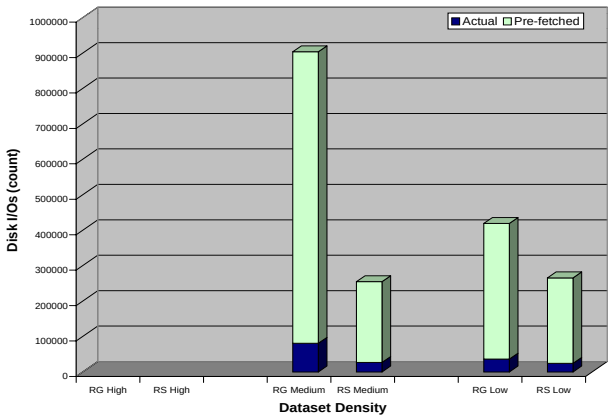
(c) Timings for top-1000 queries (0.10% selectivity)



(d) Disk I/Os for top-1000 queries (0.10% selectivity)



(e) Timings for top-10000 queries (1.00% selectivity)



(f) Disk I/Os for top-10000 queries (1.00% selectivity)

Figure 4: Run time and disk I/O comparisons for the Repeated Guess (RG) and Ranked Sort (RS) methods.

We also note that the number of logical page requests is consistently reduced in our approach. However, due to the fact that the search is done in a spiral fashion, these page requests are more random and lead to a lower cache hit rate. Still, despite of the lower hit rate, our approach has comparable I/O performance and a lower overall running time. We should note the amount of random I/O can be reduced significantly simply by sorting all record IDs before doing the actual record fetch. Record ID sorting and prefetching will increase the cache hit rate considerably, and given the overall reduction in disk page requests of the proposed approach, it would likely lead to significant further improvements in I/O times.

6 Conclusions and Future Work

In this paper we have introduced the notion of aggregate predicates, or predicates whose value depends on a set of tuples, and we addressed the problem of search based upon aggregate predicates. We identified ranked search, or top-k queries, as an instance of the above problem in many multimedia applications, geographic information systems, as well as traditional databases with structured data. We then proposed a method for incorporating aggregate predicates and ranked search into the database engine. More specifically, we defined the syntax and explained the semantics of aggregate predicates, thus enhancing the expressive power of the SQL query language and improving usability. The added functionality allows for very powerful queries to be executed as a single SQL statement, with the potential to achieve significant performance gains. We also provided a mechanism for natively supporting aggregate predicates and treating them uniformly with scalar predicates with respect to indexing and query optimization purposes. Our proposed framework offers performance improvement for many applications by introducing a new rk_SORT operator that limits the search space and allows early termination while guaranteeing correctness of the output. To the best of our knowledge, our framework is the first to natively support aggregate predicates with the ability to exploit user-defined indexes. The effectiveness of the proposed method is corroborated by the empirical results, and overall, the usability and performance advantages of our method enable the use of databases in a wider variety of applications.

As future work, it would be interesting to explore ways of supporting approximate top-k queries and ranked search based on approximate evaluation of aggregate predicates. In many applications, the scoring function used for ranking can be approximated much more efficiently than having it evaluated exactly. In some of those applications, it is sufficient to answer a top-k query approximately by returning tuples that are within a certain distance to the true top k tuples.

References

Aoki, P. M. (1997), Generalizing "search" in generalized search trees, Technical Report UCB//CSD-97-950, University of California, Berkeley, CA.

Carey, M. & Kossmann, D. (1998), Reducing the braking distance of an SQL query engine, *in* 'Proc. of the 1998 Intl. Conference on Very Large Databases'.

Chakrabarti, K. & Mehrotra, S. (1998), The hybrid tree: An index structure for indexing high dimensional feature spaces, Technical Report MARS-TR-99-01, University of California, Irvine, CA. Extended Version.

Chaudhuri, S. & Gravano, L. (1999), Evaluating top-k selection queries, *in* 'Proc. of the 25th Intl. Conference on Very Large Databases (VLDB '98)', Edinburgh, Scotland, pp. 399–410.

Chen, W., Chow, J.-H., Fuh, Y.-C., Grandbois, J., Jou, M., Mattos, N., Tran, B. & Wang, Y. (1999), High level indexing of user-defined types, *in* 'Proc. of the 25th Intl. Conference on Very Large Databases (VLDB '98)', Edinburgh, Scotland, pp. 554–564.

Donjerkovic, D. & Ramakrishnan, R. (1999), Probabilistic optimization of top N queries, *in* 'Proc. of the 25th Intl. Conference on Very Large Databases', Edinburgh, Scotland, pp. 411–422.

Fagin, R. (1998), Fuzzy queries in multimedia database systems, *in* 'Proc. of the 1998 ACM Symposium on Principles of Database Systems'.

Hass, L. M., Chang, W., Lohman, G. M., McPherson, J., Wilms, P. F., Lapis, G., Lindsay, B., Pirahesh, H., Carey, M. & Shekia, E. (1990), 'Starburst mid-flight: As the dust clears', *IEEE Transactions on Knowledge and Data Engineering* pp. 143–160.

Hellerstein, J. M., Naughton, J. F. & Pfeiffer, A. (1995), Generalized search trees for database systems, *in* 'Proc. of the 21st Intl. Conference on Very Large Databases (VLDB '95)', Zurich, Switzerland.

Kossmann, D. & Carey, M. (1997), On saying "enough already!" in SQL, *in* 'Proc. of the 1997 ACM-SIGMOD Conference on Management of Data', Tucson, Arizona.

Roussopoulos, N., Kelley, S. & Vincent, F. (1995), Nearest neighbor queries, *in* 'Proc. of the 1995 ACM Intl. Conference on Management of Data (SIGMOD '95)', San Jose, CA.

Sellinger, P. G., Astrahan, M., Chamberlin, D., Lorie, R. & Price, T. (1979), Access path selection in a relational database management system, *in* 'Proc. ACM-SIGMOD Intl. Conference on Management of Data', Boston, MA.

Stonebraker, M. R., Wong, E. & Kreps, P. (1976), 'The design and implementation of INGRES', *ACM Transactions on Database Systems* **1**(3), 189–222.

White, D. A. & Jain, R. (1996), Similarity indexing with the SS-tree, *in* 'Proc. of the 12th IEEE Intl. Conference on Data Engineering', pp. 516–523.