A Survey of Top-k Query Processing Techniques in Relational Database Systems

IHAB F. ILYAS, GEORGE BESKALES, and MOHAMED A. SOLIMAN

University of Waterloo

Efficient processing of top-k queries is a crucial requirement in many interactive environments that involve massive amounts of data. In particular, efficient top-k processing in domains such as the Web, multimedia search, and distributed systems has shown a great impact on performance. In this survey, we describe and classify top-k processing techniques in relational databases. We discuss different design dimensions in the current techniques including query models, data access methods, implementation levels, data and query certainty, and supported scoring functions. We show the implications of each dimension on the design of the underlying techniques. We also discuss top-k queries in XML domain, and show their connections to relational approaches.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Top-k, rank-aware processing, rank aggregation, voting

ACM Reference Format:

Ilyas, I. F., Beskales, G., and Soliman, M. A. 2008. A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. 40, 4, Article 11 (October 2008), 58 pages DOI = 10.1145/1391729.1391730 http://doi.acm.org/10.1145/1391729.1391730

1. INTRODUCTION

Information systems of different types use various techniques to rank query answers. In many application domains, end-users are more interested in the most important (top-k) query answers in the potentially huge answer space. Different emerging applications warrant efficient support for top-k queries. For instance, in the context of the Web, the effectiveness and efficiency of metasearch engines, which combine rankings from different search engines, are highly related to efficient rank aggregation methods. Similar applications exist in the context of information retrieval [Salton and McGill 1983] and data mining [Getoor and Diehl 2005]. Most of these applications compute queries that involve joining and aggregating multiple inputs to provide users with the top-k results.

Support was provided in part by the Natural Sciences and Engineering Research Council of Canada through Grant 311671-05.

Authors' Address: University of Waterloo, 200 University Ave. West, Waterloo, ON, Canada N2L 3G1; email: {ilyas,gbeskale,m2ali}@uwaterloo.ca.

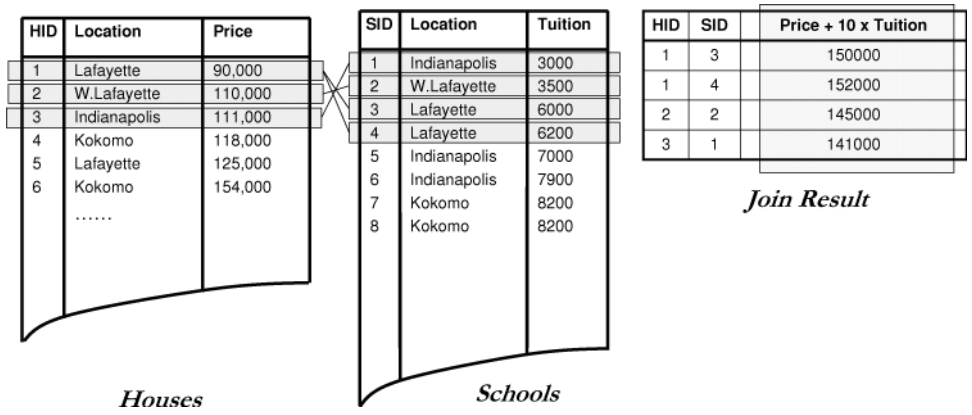

Fig. 1. A top-k query example.

One common way to identify the top-k objects is scoring all objects based on some *scoring function*. An object score acts as a valuation for that object according to its characteristics (e.g., price and size of house objects in a real estate database, or color and texture of images in a multimedia database). Data objects are usually evaluated by multiple scoring predicates that contribute to the total object score. A scoring function is therefore usually defined as an aggregation over partial scores.

Top-k processing connects to many database research areas including query optimization, indexing methods, and query languages. As a consequence, the impact of efficient top-k processing is becoming evident in an increasing number of applications. The following examples illustrate real-world scenarios where efficient top-k processing is crucial. The examples highlight the importance of adopting efficient top-k processing techniques in traditional database environments.

Example 1.1. Consider a user interested in finding a location (e.g., city) where the combined cost of buying a house and paying school tuition for 10 years at that location is minimum. The user is interested in the five least expensive places. Assume that there are two external sources (databases), Houses and Schools, that can provide information on houses and schools, respectively. The Houses database provides a ranked list of the cheapest houses and their locations. Similarly, the Schools database provides a ranked list of the least expensive schools and their locations. Figure 1 gives an example of the Houses and Schools databases.

A naïve way to answer the query described in Example 1.1 is to retrieve two lists: a list of the cheapest houses from Houses, and a list of the cheapest schools from Schools. These two lists are then joined based on location such that a valid join result is comprised of a house and a school at the same location. For all join results, the total cost of each house-school pair is computed, for example, by adding the house price and the school tuition for 10 years. The five cheapest pairs constitute the final answer to this query. Figure 1 shows an illustration for the join process between houses and schools lists, and partial join results. Note that the top five results cannot be returned to the user until all the join results are generated. For large numbers of colocated houses and schools, the processing of such a query, in the traditional manner, is very expensive as it requires expensive join and sort operations for large amounts of data.

Example 1.2. Consider a video database system where several visual features are extracted from each video object (frame or segment). Example features include color

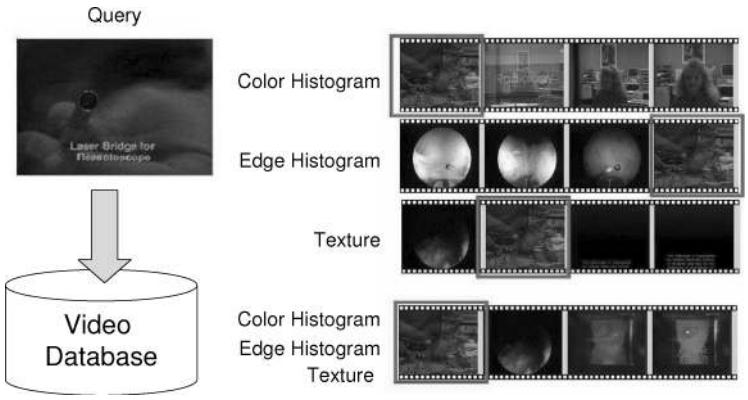

Fig. 2. Single and multifeature queries in video database.

histograms, color layout, texture, and edge orientation. Features are stored in separate relations indexed using high-dimensional indexes that support similarity queries. Suppose that a user is interested in the top 10 video frames most similar to a given query image based on a set of visual features.

Example 1.2 draws attention to the importance of efficient top-k processing in similarity queries. In video databases [Aref et al. 2004], hours of video data are stored inside the database producing huge amounts of data. Top-k similarity queries are traditionally answered using high-dimensional indexes built on individual video features, and a nearest-neighbor scan operator on top of these indexes. A database system that supports approximate matching ranks objects depending on how well they match the query example. Figure 2 presents an example of single-feature similarity query based on color histogram, texture, and edge orientation. More useful similarity queries could involve multiple features. For example, suppose that a user is interested in the top 10 video frames most similar to a given query image based on color and texture combined. The user could provide a function that combines similarity scores in both features into an overall similarity score. For example, the global score of a frame f with respect to a query image q could be computed as $0.5 \times \mathit{ColorSimilarity}(f, q) + 0.5 \times \mathit{TextureSimilarity}(f, q)$.

One way to answer such a multifeature query is by sequentially scanning all database objects, computing the score of each object according to each feature, and combining the scores into a total score for each object. This approach suffers from scalability problems with respect to database size and the number of features. An alternative way is to map the query into a join query that joins the output of multiple single-feature queries, and then sorts the joined results based on combined score. This approach also does not scale with respect to both number of features and database size since all join results have to be computed then sorted.

The main problem with sort-based approaches is that sorting is a blocking operation that requires full computation of the join results. Although the input to the join operation is sorted on individual features, this order is not exploited by conventional join algorithms. Hence, sorting the join results becomes necessary to produce the top-k answers. Embedding rank-awareness in query processing techniques provides a more efficient and scalable solution.

In this survey, we discuss the state-of-the-art top-k query processing techniques in relational database systems. We give a detailed coverage for most of the recently presented techniques focusing primarily on their integration into relational database

Table I. Frequently Used Notations

Notation	Description
m	Number of sources (lists)
L_i	Ranked source (list) number i
t or o	A tuple or object to be scored
g	A group of tuples based on some grouping attributes
F	Scoring (ranking) Function
$\underline{F}(t)$ or $\underline{F}(o)$	Score lower bound of t (or o)
$\overline{F}(t)$ or $\overline{F}(o)$	Score upper bound of t (or o)
$p_i(t)$ or $p_i(o)$	The value of scoring predicate p_i applied to t (or o); predicate p_i determines objects order in L_i
p_i^{max}	The maximum score of predicate p_i
p_i^{min}	The minimum score of predicate p_i
$\overline{p}_i$	The score upper bound of predicate p_i (mostly refers to the score of the last seen object in L_i)
T	Score threshold (cutoff value)
A_k	The current top-k set
M_k	The minimum score in the current top-k set

environments. We also introduce a taxonomy to classify top-k query processing techniques based on multiple design dimensions, described in the following:

—*Query model*. Top-k processing techniques are classified according to the query model they assume. Some techniques assume a selection query model, where scores are attached directly to base tuples. Other techniques assume a join query model, where scores are computed over join results. A third category assumes an aggregate query model, where we are interested in ranking groups of tuples.

—*Data access methods*. Top-k processing techniques are classified according to the data access methods they assume to be available in the underlying data sources. For example, some techniques assume the availability of random access, while others are restricted to only sorted access.

—*Implementation level*. Top-k processing techniques are classified according to their level of integration with database systems. For example, some techniques are implemented in an application layer on top of the database system, while others are implemented as query operators.

—*Data and query uncertainty*. Top-k processing techniques are classified based on the uncertainty involved in their data and query models. Some techniques produce exact answers, while others allow for approximate answers, or deal with uncertain data.

—*Ranking function*. Top-k processing techniques are classified based on the restrictions they impose on the underlying ranking (scoring) function. Most proposed techniques assume monotone scoring functions. Few proposals address general functions.

1.1. Notations

The working environments, of most of the techniques we describe, assume a scoring (ranking) function used to score objects (tuples) by aggregating the values of their partial scores (scoring predicates). Table I lists the frequently used notations in this survey.

1.2. Outline

The remainder of this survey is organized as follows. Section 2 introduces the taxonomy we adopt in this survey to classify top-k query processing methods. Sections 3, 4, 5, and 6 discuss the different design dimensions of top-k processing techniques, and give the details of multiple techniques in each dimension. Section 7 discusses related top-k

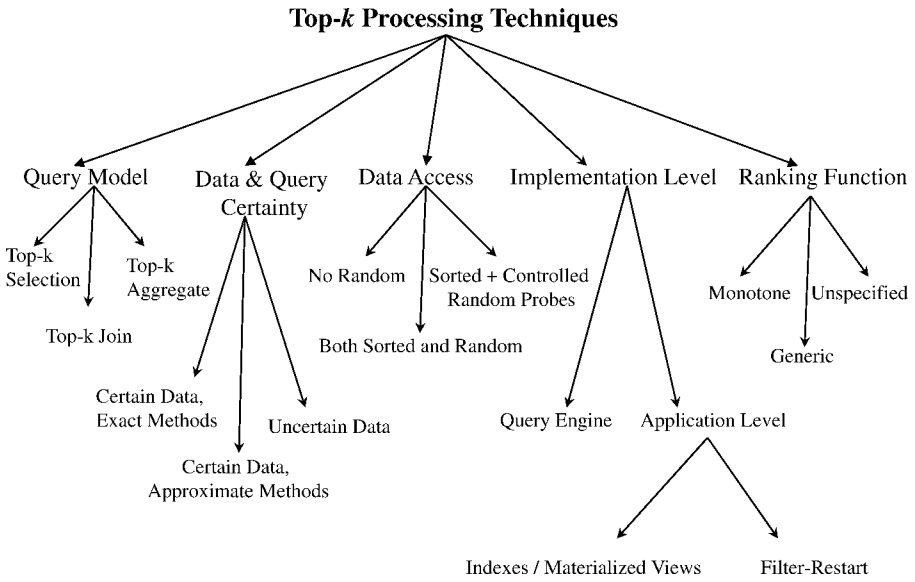

Fig. 3. Classification of top-k query processing techniques.

processing techniques for XML data. Section 8 presents related background from voting theory, which forms the basis of many current top-k processing techniques. Section 9 concludes this survey, and describes future research directions.

We assume the reader of this survey has a general familiarity with relational database concepts.

2. TAXONOMY OF TOP-*K* QUERY PROCESSING TECHNIQUES

Supporting efficient top-k processing in database systems is a relatively recent and active line of research. Top-k processing has been addressed from different perspectives in the current literature. Figure 3 depicts the classification we adopt in this survey to categorize different top-k processing techniques based on their capabilities and assumptions. In the following sections, we discuss our classification dimensions, and their impact on the design of the underlying top-k processing techniques. For each dimension, we give a detailed description for one or more example techniques.

2.1. Query Model Dimension

Current top-k processing techniques adopt different query models to specify the data objects to be scored. We discuss three different models: (1) top-k selection query, (2) top-k join query, and (3) top-k aggregate query. We formally define these query models in the following.

2.1.1. Top-k Selection Query Model. In this model, the scores are assumed to be attached to base tuples. A top-k selection query is required to report the k tuples with the highest scores. Scores might not be readily available since they could be the outcome of some user-defined scoring function that aggregates information coming from different tuple attributes.

Definition 2.1 (*Top-k Selection Query*). Consider a relation R, where each tuple in R has n attributes. Consider m scoring predicates, $p_1 \cdots p_m$ defined on these attributes.

Let $F(t) = F(p_1(t), \ldots, p_m(t))$ be the overall score of tuple $t \in R$. A top-k selection query selects the k tuples in R with the largest F values.

A SQL template for top-k selection query is the following:

```
SELECT some_attributes
FROM R
WHERE selection_condition
ORDER BY F(p_1, ..., p_m)
LIMIT k[1]
```

Consider Example 1.2. Assume the user is interested in finding the top-10 video objects that are most similar to a given query image q, based on color and texture, and whose release date is after 1/1/2008. This query could be written as follows:

```
SELECT v.id
FROM VideoObject v
WHERE v.date > '01/01/2008'
ORDER BY 0.5 * Color Similarity(q, v) + 0.5 * TextureSimilarity(q, v)
LIMIT 10
```

The NRA algorithm [Fagin et al. 2001] is one example of top-k techniques that adopt the top-k selection model. The input to the NRA algorithm is a set of sorted lists; each ranks the "same" set of objects based on different attributes. The output is a ranked list of these objects ordered on the aggregate input scores. We give the full details of this algorithm in Section 3.2.

2.1.2. Top-k Join Query Model. In this model, scores are assumed to be attached to join results rather than base tuples. A top-k join query joins a set of relations based on some arbitrary join condition, assigns scores to join results based on some scoring function, and reports the top-k join results.

Definition 2.2 (*Top-k Join Query*). Consider a set of relations $R_1 \cdots R_n$. A top-k join query joins $R_1 \cdots R_n$, and returns the k join results with the largest combined scores. The combined score of each join result is computed according to some function $F(p_1, \ldots, p_m)$, where $p_1, \ldots, p_m$ are scoring predicates defined over the join results.

A possible SQL template for a top-k join query is

```
SELECT *
FROM R_1, ..., R_n
WHERE join_condition(R_1, ..., R_n)
ORDER BY F(p_1, ..., p_m)
LIMIT k
```

For example, the top-k join query in Example 1.1 could be formulated as follows:

```
SELECT h.id, s.id
FROM House h, School s
WHERE h.location=s.location
ORDER BY h.price + 10 * s.tuition
LIMIT 5
```

A top-k selection query can be formulated as a special top-k join query by partitioning R into n vertical relations $R_1, \ldots, R_n$, such that each relation R_i has the necessary attributes to compute the score p_i. For example, Let R contains the attributes tid, A_1, A_2, and A_3. Then, R can be partitioned into $R_1 = (tid, A_1)$ and $R_2 = (tid, A_2, A_3)$, where

[1]Other keywords, for example, Stop After k, are also used in other SQL dialects.

$p_1 = A_1$ and $p_2 = A_2 + A_3$. In this case, the join condition is an equality condition on key attributes. The NRA-RJ algorithm [Ilyas et al. 2002] is one example of top-k processing techniques that formulate top-k selection queries as top-k join queries based on tuples' keys.

Many top-k join techniques address the interaction between computing the join results and producing the top-k answers. Examples are the J^* algorithm [Natsev et al. 2001] (Section 3.2), and the Rank-Join algorithm [Ilyas et al. 2004a] (Section 4.2). Some techniques, for example, PREFER [Hristidis et al. 2001] (Section 4.1.2), process top-k join queries using auxiliary structures that materialize join results, or by ranking the join results after they are generated.

2.1.3. Top-k Aggregate Query Model. In this model, scores are computed for tuple groups, rather than individual tuples. A top-k aggregate query reports the k groups with the largest scores. Group scores are computed using a group aggregate function such as *sum*.

Definition 2.3 (*Top-k Aggregate Query*). Consider a set of grouping attributes $\mathcal{G}$=$\{g_1, \ldots, g_r\}$, and an aggregate function F that is evaluated on each group. A top-k aggregate query returns the k groups, based on $\mathcal{G}$, with the highest F values.

A SQL formulation for a top-k aggregate query is

```
SELECT  g_1, ..., g_r, F
FROM  R_1, ..., R_n
WHERE  join_condition(R_1, ..., R_n)
GROUP BY  g_1, ..., g_r
ORDER BY  F
LIMIT  k
```

An example top-k aggregate query is to find the best five areas to advertise student insurance product, based on the score of each area, which is a function of student's income, age, and credit.

```
SELECT  zipcode,Average(income*w1 + age*w2 + credit*w3) as score
FROM  customer
WHERE  occupation = 'student'
GROUP  BY zipcode
ORDER BY  score
LIMIT 5
```

Top-k aggregate queries add additional challenges to top-k join queries: (1) interaction of grouping, joining, and scoring of query results, and (2) nontrivial estimation of the scores of candidate top-k groups. A few recent techniques, for example, Li et al. [2006], address these challenges to efficiently compute top-k aggregate queries. We discuss these techniques in Section 4.2.

2.2. Data Access Dimension

Many top-k processing techniques involve accessing multiple data sources with different valuations of the underlying data objects. A typical example is a metasearcher that aggregates the rankings of search hits produced by different search engines. The hits produced by each search engine can be seen as a ranked list of Web pages based on some score, for example, relevance to query keywords. The manner in which these lists are accessed largely affects the design of the underlying top-k processing techniques. For example, ranked lists could be scanned sequentially in score order. We refer to this access method as *sorted access*. Sorted access is supported by a DBMS if, for example,

a B-Tree index is built on objects' scores. In this case, scanning the sequence set (leaf level) of the B-Tree index provides a sorted access of objects based on their scores. On the other hand, the score of some object might be required directly without traversing the objects with higher/smaller scores. We refer to this access method as *random access*. Random access could be provided through index lookup operations if an index is built on object keys.

We classify top-k processing techniques, based on the assumptions they make about available data access methods in the underlying data sources, as follows:

—*Both sorted and random access*. In this category, top-k processing techniques assume the availability of both sorted and random access in all the underlying data sources. Examples are TA [Fagin et al. 2001], and the Quick-Combine algorithm [Güntzer et al. 2000]. We discuss the details of these techniques in Section 3.1.

—*No random access*. In this category, top-k processing techniques assume the underlying sources provide only sorted access to data objects based on their scores. Examples are the NRA algorithm [Fagin et al. 2001], and the Stream-Combine algorithm [Güntzer et al. 2001]. We discuss the details of these techniques in Section 3.2.

—*Sorted access with controlled random probes*. In this category, top-k processing techniques assume the availability of at least one sorted access source. Random accesses are used in a controlled manner to reveal the overall scores of candidate answers. Examples are the Rank-Join algorithm [Ilyas et al. 2004a], the MPro algorithm [Chang and Hwang 2002], and the Upper and Pick algorithms [Bruno et al. 2002b]. We discuss the details of these techniques in Section 3.3.

2.3. Implementation-Level Dimension

Integrating top-k processing with database systems is addressed in different ways by current techniques. One approach is to embed top-k processing in an outer layer on top of the database engine. This approach allows for easy extensibility of top-k techniques, since they are decoupled from query engines. The capabilities of database engines (e.g., storage, indexing, and query processing) are leveraged to allow for efficient top-k processing. New data access methods or specialized data structures could also be built to support top-k processing. However, the core of query engines remains unchanged.

Another approach is to modify the core of query engines to recognize the ranking requirements of top-k queries during query planning and execution. This approach has a direct impact on query processing and optimization. Specifically, query operators are modified to be rank-aware. For example, a join operator is required to produce ranked join results to support pipelining top-k query answers. Moreover, available access methods for ranking predicates are taken into account while optimizing a query plan.

We classify top-k processing techniques based on their level of integration with database engines as follows:

—*Application level*. This category includes top-k processing techniques that work outside the database engine. Some of the techniques in this category rely on the support of specialized top-k indexes or materialized views. However, the main top-k processing remains outside the engine. Examples are Chang et al. [2000], and Hristidis et al. [2001]. Another group of techniques formulate top-k queries as range queries that are repeatedly executed until the top-k objects are obtained. We refer to this group of techniques as *filter-restart*. One example is Donjerkovic and Ramakrishnan [1999]. We discuss the details of these techniques in Section 4.1.

—*Query engine level*. This category includes techniques that involve modifications to the query engine to allow for rank-aware processing and optimization. Some of these techniques introduce new query operators to support efficient top-k processing. For example, Ilyas et al. [2004a] introduced rank-aware join operators. Other techniques, for example, Li et al. [2005, 2006], extend rank-awareness to query algebra to allow for extensive query optimization. We discuss the details of these techniques in Section 4.2.

2.4. Query and Data Uncertainty Dimension

In some query processing environments, for example, decision support or OLAP, obtaining exact query answers efficiently may be overwhelming to the database engine because of the interactive nature of such environments, and the sheer amounts of data they usually handle. Such environments could sacrifice the accuracy of query answers in favor of performance. In these settings, it may be acceptable for a top-k query to report approximate answers.

The uncertainty in top-k query answers might alternatively arise due to the nature of the underlying data itself. Applications in domains such as sensor networks, data cleaning, and moving objects tracking involve processing data that is probabilistic in nature. For example, the temperature reading of some sensor could be represented as a probability distribution over a continuous interval, or a customer name in a dirty database could be a represented as a set of possible names. In these settings, top-k queries, as well as other query types, need to be formulated and processed while taking data uncertainty into account.

We classify top-k processing techniques based on query and data certainty as follows:

—*Exact methods over certain data*. This category includes the majority of current top-k processing techniques, where deterministic top-k queries are processed over deterministic data.

—*Approximate methods over certain data*. This category includes top-k processing techniques that operate on deterministic data, but report approximate answers in favor of performance. The approximate answers are usually associated with probabilistic guarantees indicating how far they are from the exact answer. Examples include Theobald et al. [2005] and Amato et al. [2003]. We discuss the details of these techniques in Section 5.1.

—*Uncertain data*. This category includes top-k processing techniques that work on probabilistic data. The research proposals in this category formulate top-k queries based on different uncertainty models. Some approaches treat probabilities as the only scoring dimension, where a top-k query is a Boolean query that reports the k most probable query answers. Other approaches study the interplay between the scoring and probability dimensions. Examples are Ré et al. [2007] and Soliman et al. [2007]. We discuss the details of these techniques in Section 5.2.

2.5. Ranking Function Dimension

The properties of the ranking function largely influence the design of top-k processing techniques. One important property is the ability to upper bound objects' scores. This property allows early pruning of certain objects without exactly knowing their scores. A *monotone* ranking function can largely facilitate upper bound computation. A function F, defined on predicates $p_1, \ldots, p_n$, is monotone if $F(p_1, \ldots, p_n) \leq F(p'_1, \ldots, p'_n)$

whenever $p_i \leq \acute{p}_i$ for every i. We elaborate on the impact of function monotonicity on top-k processing in Section 6.1.

In more complex applications, a ranking function might need to be expressed as a numeric expression to be optimized. In this setting, the monotonicity restriction of the ranking function is relaxed to allow for more generic functions. Numerical optimization tools as well as indexes are used to overcome the processing challenges imposed by such ranking functions.

Another group of applications address ranking objects without specifying a ranking function. In some environments, such as data exploration or decision making, it might not be important to rank objects based on a specific ranking function. Instead, objects with high quality based on different data attributes need to be reported for further analysis. These objects could possibly be among the top-k objects of some unspecified ranking function. The set of objects that are not *dominated* by any other objects, based on some given attributes, are usually referred to as the *skyline*.

We classify top-k processing techniques based on the restrictions they impose on the underlying ranking function as follows:

—*Monotone ranking function*. Most of the current top-k processing techniques assume monotone ranking functions since they fit in many practical scenarios, and have appealing properties allowing for efficient top-k processing. One example is Fagin et al. [2001]. We discuss the properties of monotone ranking functions in Section 6.1.

—*Generic ranking function*. A few recent techniques, for example, Zhang et al. [2006], address top-k queries in the context of constrained function optimization. The ranking function in this case is allowed to take a generic form. We discuss the details of these techniques in Section 6.2.

—*No ranking function*. Many techniques have been proposed to answer skyline-related queries, for example, Börzsönyi et al. [2001] and Yuan et al. [2005]. Covering current skyline literature in detail is beyond the scope of this survey. We believe it worth a dedicated survey by itself. However, we briefly show the connection between *skyline* and top-k queries in Section 6.3.

2.6. Impact of Design Dimensions on Top-*k* Processing Techniques

Figure 4 shows the properties of a sample of different top-k processing techniques that we describe in this survey. The applicable categories under each taxonomy dimension are marked for each technique. For example, TA [Fagin et al. 2001] is an *exact* method that assumes top-k selection query model, and operates on *certain data*, exploiting both *sorted and random* access methods. TA integrates with database systems at the *application level*, and supports *monotone* ranking functions.

Our taxonomy encapsulates different perspectives to understand the processing requirements of current top-k processing techniques. The taxonomy dimensions, discussed in the previous sections, can be viewed as design dimensions that impact the capabilities and the assumptions of the underlying top-k algorithms. In the following, we give some examples of the impact of each design dimension on the underlying top-k processing techniques:

—*Impact of query model.* The query model significantly affects the solution space of the top-k algorithms. For example, the top-k join query model (Definition 2.2) imposes tight integration with the query engine and physical join operators to efficiently navigate the Cartesian space of join results.

—*Impact of data access.* Available access methods affect how different algorithms compute bounds on object scores and hence affect the termination condition. For example,

	Query model			Data & query certainty			Data access			Implement. level		Ranking function	
	Top-k selection	Top-k join	Top-k aggregate	Certain data, exact methods	Certain data, approx. methods	Uncertain data	No random	Both sorted and random	Sorted + controlled random probes	Query engine level	Application level	Monotone	Generic
TA [Fagin et al. 2001], Quick-Combine [Güntzer et al. 2000]	✓			✓				✓			✓	✓	
TA-Θ approx [Fagin et al. 2003]	✓				✓			✓			✓	✓	
NRA [Fagin et al. 2001], Stream-Combine [Güntzer et al. 2001]	✓			✓			✓				✓	✓	
CA [Fagin et al. 2001]	✓			✓				✓			✓	✓	
Upper/Pick [Bruno et al. 2002b]	✓			✓					✓		✓	✓	
Mpro [Chang and Hwang 2002]	✓			✓					✓		✓	✓	
J* [Natsev et al. 2001]		✓		✓			✓				✓	✓	
J* e-approx. [Natsev et al. 2001]		✓			✓		✓				✓	✓	
PREFER [Hristidis et al. 2001], Filter-Restart [Bruno et al. 2002a], Onion Indices [Chang et al. 2000], LPTA [Das et al. 2006]	✓			✓			N/A				✓	✓	
NRA-RJ [Ilyas et al. 2002]	✓			✓			✓			✓		✓	
Rank-Join [Ilyas et al. 2003]		✓		✓					✓	✓		✓	
RankSQL-μ operator [Li et al. 2005]	✓			✓					✓	✓		✓	
rankaggr Operator [Li et al. 2006]			✓	✓			✓			✓		✓	
TopX [Theobald et al. 2005]	✓				✓			✓			✓	✓	
KLEE [Michel et al. 2005]	✓				✓		✓				✓	✓	
OPT* [Zhang et al. 2006]		✓		✓				N/A			✓		✓
OPTU-Topk [Soliman et al. 2007]		✓				✓	✓				✓	✓	
MS_Topk [Ré et al. 2007]		✓				✓		N/A			✓	✓	

Fig. 4. Properties of different top-k processing techniques.

the NRA algorithm [Fagin et al. 2001], discussed in Section 3.2, has to compute a "range" of possible scores for each object since the lack of random access prevents computing an exact score for each seen object. On the other hand, allowing random access to the underlying data sources triggers the need for cost models to optimize the number of random and sorted accesses. One example is the CA algorithm [Fagin et al. 2001], discussed in Section 3.1.

—*Impact of data and query uncertainty.* Supporting approximate query answers requires building probabilistic models to fit the score distributions of the underlying data, as proposed in Theobald et al. [2004] (Section 5.1.2). Uncertainty in the underlying data adds further significant computational complexity because of the huge space of possible answers that needs to be explored. Building efficient search algorithms to explore such space is crucial, as addressed in Soliman et al. [2007].

—*Impact of implementation level.* The implementation level greatly affects the requirements of the top-k algorithm. For example, implementing top-k pipelined query operator necessitates using algorithms that require no random access to their inputs to fit in pipelined query models; it also requires the output of the top-k algorithm to be a valid input to another instance of the algorithm [Ilyas et al. 2004a]. On the other hand, implementation on the application level does not have these requirements. More details are given in Section 4.2.1.

—*Impact of ranking function.* Assuming monotone ranking functions allows top-k processing techniques to benefit from the monotonicity property to guarantee early-out of query answers. Dealing with nonmonotone functions requires more sophisticated bounding for the scores of unexplored answers. Existing indexes in the database are currently used to provide such bounding, as addressed in Xin et al. [2007] (Section 6).

3. DATA ACCESS

In this section, we discuss top-k processing techniques that make different assumptions about available access methods supported by data sources. The primary data access methods are *sorted access*, *random access*, and a combination of both methods. In sorted access, objects are accessed sequentially ordered by some scoring predicate, while for random access, objects are directly accessed by their identifiers.

The techniques presented in this section assume multiple lists (possibly located at separate sources) that rank the same set of objects based on different scoring predicates. A score aggregation function is used to aggregate partial objects' scores, obtained from the different lists, to find the top-k answers.

The cost of executing a top-k query, in such environments, is largely influenced by the supported data access methods. For example, random access is generally more expensive than sorted access. A common assumption in all of the techniques discussed in this section is the existence of at least one source that supports sorted access. We categorize top-k processing techniques, according to the assumed source capabilities, into the three categories described in the next sections.

3.1. Both Sorted and Random Access

Top-k processing techniques in this category assume data sources that support both access methods, sorted and random. Random access allows for obtaining the overall score of some object right after it appears in one of the data sources. The *Threshold Algorithm (TA)* and *Combined Algorithm (CA)* [Fagin et al. 2001] belong to this category.

Algorithm 1 describes the details of TA. The algorithm scans multiple lists, representing different rankings of the same set of objects. An upper bound T is maintained for the overall score of unseen objects. The upper bound is computed by applying the scoring function to the partial scores of the last seen objects in different lists. Notice that the last seen objects in different lists could be different. The upper bound is updated every time a new object appears in one of the lists. The overall score of some seen object is computed by applying the scoring function to object's partial scores, obtained from different lists. To obtain such partial scores, each newly seen object in one of the lists is looked up in all other lists, and its scores are aggregated using the scoring function to obtain the overall score. All objects with total scores that are greater than or equal to T can be reported. The algorithm terminates after returning the kth output. Example 3.1 illustrates the processing of TA.

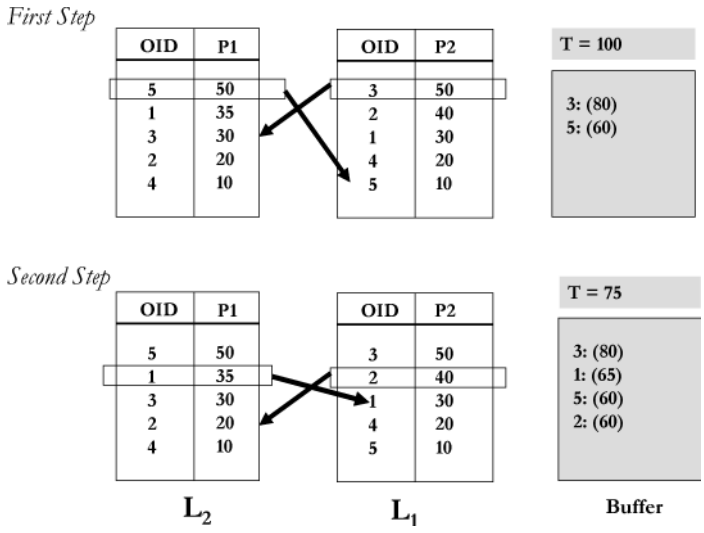

Fig. 5. The Threshold Algorithm (TA).

Algorithm 1. TA [Fagin et al. 2001]

---

(1) Do sorted access in parallel to each of the m sorted lists L_i. As a new object o is seen under sorted access in some list, do random access to the other lists to find $p_i(o)$ in every other list L_i. Compute the score $F(o) = F(p_1, \ldots, p_m)$ of object o. If this score is among the k highest scores seen so far, then remember object o and its score $F(o)$ (ties are broken arbitrarily, so that only k objects and their scores are remembered at any time).

(2) For each list L_i, let $\overline{p}_i$ be the score of the last object seen under sorted access. Define the *threshold value* T to be $F(\overline{p}_1, \ldots, \overline{p}_m)$. As soon as at least k objects have been seen with scores at least equal to T, halt.

(3) Let A_k be a set containing the k seen objects with the highest scores. The output is the sorted set $\{(o, F(o))|o \in A_k\}$.

Example 3.1 (*TA Example*). Consider two data sources L_1 and L_2 holding different rankings for the same set of objects based on two different scoring predicates p_1 and p_2, respectively. Each of p_1 and p_2 produces score values in the range [0, 50]. Assume each source supports sorted and random access to their ranked lists. Consider a score aggregation function $F = p_1 + p_2$. Figure 5 depicts the first two steps of TA. In the first step, retrieving the top object from each list, and probing the value of its other scoring predicate in the other list, result in revealing the exact scores for the top objects. The seen objects are buffered in the order of their scores. A threshold value, T, for the scores of unseen objects is computed by applying F to the last seen scores in both lists, which results in $50 + 50 = 100$. Since both seen objects have scores less than T, no results can be reported. In the second step, T drops to 75, and object 3 can be safely reported since its score is above T. The algorithm continues until k objects are reported, or sources are exhausted.

TA assumes that the costs of different access methods are the same. In addition, TA does not have a restriction on the number of random accesses to be performed. Every sorted access in TA results in up to $m - 1$ random accesses, where m is the number of lists. Such a large number of random accesses might be very expensive. The CA

algorithm [Fagin et al. 2001] alternatively assumes that the costs of different access methods are different. The CA algorithm defines a ratio between the costs of the two access methods to control the number of random accesses, since they usually have higher costs than sorted accesses.

The CA algorithm periodically performs random accesses to collect unknown partial scores for objects with the highest score lower bounds (ties are broken using score upper bounds). A score lower bound is computed by applying the scoring function to object's known partial scores, and the *worst* possible unknown partial scores. On the other hand, a score upper bound is computed by applying the scoring function to object's known partial scores, and the *best* possible unknown partial scores. The *worst* unknown partial scores are the lowest values in the score range, while the *best* unknown partial scores are the last seen scores in different lists. One random access is performed periodically every Δ sorted accesses, where Δ is the floor of the ratio between random access cost and sorted access cost.

Although CA minimizes the number of random accesses compared to TA, it assumes that all sources support random access at the same cost, which may not be true in practice. This problem is addressed in Bruno et al. [2002b] and Marian et al. [2004], and we discuss it in more detail in Section 3.3.

In TA, tuples are retrieved from sorted lists in a round-robin style. For instance, if there are m sorted access sources, tuples are retrieved from sources in this order: $(L_1, L_2, \ldots, L_m, L_1, \ldots)$. Two observations can possibly minimize the number of retrieved tuples. First, sources with rapidly decreasing scores can help decrease the upper bound of unseen objects' scores (T) at a faster rate. Second, favoring sources with considerable influence on the overall scores could lead to identifying the top answers quickly. Based on these two observations, a variation of TA, named *Quick-Combine*, is introduced in Güntzer et al. [2000]. The *Quick-Combine* algorithm uses an indicator Δ_i, expressing the effectiveness of reading from source i, defined as follows:

$$\Delta_i = \frac{\partial F}{\partial p_i} \cdot (S_i(d_i - c) - S_i(d_i)), \tag{1}$$

where $S_i(x)$ refers to the score of the tuple at depth x in source i, and d_i is the current depth reached at source i. The rate at which score decays in source i is computed as the difference between its last seen score $S_i(d_i)$, and the score of the tuple c steps above in the ranked list, $S_i(d_i - c)$. The influence of source i on the scoring function F is captured using the partial derivative of F with respect to source's predicate p_i. The source with the maximum Δ_i is selected, at each step, to get the next object. It has been shown that the proposed algorithm is particularly efficient when the data exhibits tangible skewness.

3.2. No Random Access

The techniques we discuss in this section assume random access is not supported by the underlying sources. The *No Random Access (NRA)* algorithm [Fagin et al. 2001] and the *Stream-Combine* algorithm [Güntzer et al. 2001] are two examples of the techniques that belong to this category.

The NRA algorithm finds the top-k answers by exploiting only sorted accesses. The NRA algorithm may not report the exact object scores, as it produces the top-k answers using bounds computed over their exact scores. The score lower bound of some object t is obtained by applying the score aggregation function on t's known scores and the minimum possible values of t's unknown scores. On the other hand, the score upper bound of t is obtained by applying the score aggregation function on t's known scores

Algorithm 2. NRA [Fagin et al. 2001]

(1) Let $p_1^{min}, \ldots, p_m^{min}$ be the smallest possible values in lists $L_1, \ldots, L_m$.
(2) Do sorted access in parallel to lists $L_1, \ldots, L_m$, and at each step do the following:
   —Maintain the last seen predicate values $\overline{p}_1, \ldots, \overline{p}_m$ in the m lists.
   —For every object o with some unknown predicate values, compute a lower bound for $F(o)$, denoted $\underline{F}(o)$, by substituting each unknown predicate p_i with p_i^{min}. Similarly, Compute an upper bound $\overline{F}(o)$ by substituting each unknown predicate p_i with $\overline{p}_i$. For object o that has not been seen at all, $\underline{F}(o) = F(p_1^{min}, \ldots, p_m^{min})$, and $\overline{F}(o) = F(\overline{p}_1, \ldots, \overline{p}_m)$.
   —Let A_k be the set of k objects with the largest lower bound values $\underline{F}(.)$ seen so far. If two objects have the same lower bound, then ties are broken using their upper bounds $\overline{F}(.)$, and arbitrarily among objects that additionally tie in $\overline{F}(.)$.
   —Let M_k be the kth largest $\underline{F}(.)$ value in A_k.
(3) Call an object o *viable* if $\overline{F}(o) > M_k$. Halt when (a) at least k distinct objects have been seen, and (b) there are no viable objects outside A_k. That is, if $\overline{F}(o) \leq M_k$ for all $o \notin A_k$, return A_k.

and the maximum possible values of t's unknown scores, which are the same as the last seen scores in the corresponding ranked lists. This allows the algorithm to report a top-k object even if its score is not precisely known. Specifically, if the score lower bound of an object t is not below the score upper bounds of all other objects (including unseen objects), then t can be safely reported as the next top-k object. The details of the NRA algorithm are given in Algorithm 2.

Example 3.2 illustrates the processing of the NRA algorithm.

Example 3.2 (*NRA Example*). Consider two data sources L_1 and L_2, where each source holds a different ranking of the same set of objects based on scoring predicates p_1 and p_2, respectively. Both p_1 and p_2 produce score values in the range [0, 50]. Assume both sources support only sorted access to their ranked lists. Consider a score aggregation function $F = p_1 + p_2$. Figure 6 depicts the first three steps of the NRA algorithm. In the first step, retrieving the first object in each list gives lower and upper bounds for objects' scores. For example, object 5 has a score range of [50, 100], since the value of its known scoring predicate p_1 is 50, while the value of its unknown scoring predicate p_2 cannot exceed 50. An upper bound for the scores of unseen objects is computed as $50 + 50 = 100$, which is the result of applying F to the last seen scores in both sorted lists. The seen objects are buffered in the order of their score lower bounds. Since the score lower bound of object 5, the top buffered object, does not exceed the score upper bound of other objects, nothing can be reported. The second step adds two more objects to the buffer, and updates the score bounds of other buffered objects. In the third step, the scores of objects 1 and 3 are completely known. However, since the score lower bound of object 3 is not below the score upper bound of any other object (including the unseen ones), object 3 can be reported as the top-1 object. Note that at this step object 1 cannot be additionally reported, since the score upper bound of object 5 is 80, which is larger than the score lower bound of object 1.

The Stream-Combine algorithm [Güntzer et al. 2001] is based on the same general idea of the NRA algorithm. The Stream-Combine algorithm prioritizes reading from sorted lists to give more chance to the lists that might lead to the earliest termination. To choose which sorted list (stream) to access next, an effectiveness indicator Δ_i is computed for each stream i, similar to the Quick-Combine algorithm. The definition of Δ_i in this case captures three properties of stream i that may lead to early termination:

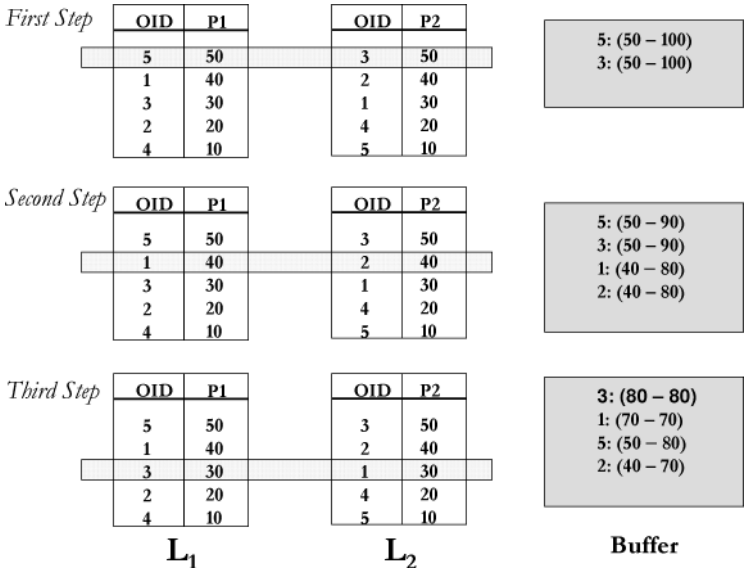

Fig. 6. The three first steps of the NRA algorithm.

(1) how rapidly scores decrease in stream i, (2) what is the influence of stream i on the total aggregated score, and (3) how many top-k objects would have their score bounds tightened by reading from stream i. The indicator Δ_i is defined as follows:

$$\Delta_i = \#M_i \cdot \frac{\partial F}{\partial p_i} \cdot (S_i(d_i - c) - S_i(d_i)) \tag{2}$$

where $S_i(x)$ refers to the score of the tuple at depth x in stream i, and d_i is the current depth reached at stream i.

The term $(S_i(d_i - c) - S_i(d_i))$ captures the rate of score decay in stream i, while the term $\frac{\partial F}{\partial p_i}$ captures how much the stream's scoring predicate contributes to the total score, similar to the Quick-Combine algorithm. The term $\#M_i$ is the number of top-k objects whose score bounds may be affected when reading from stream i, by reducing their score upper bounds, or knowing their precise scores. The stream with the maximum Δ_i is selected, at each step, to get the next object.

The NRA algorithm has been also studied in Mamoulis et al. [2006] under various application requirements. The presented techniques rely on the observation that, at some stage during NRA processing, it is not useful to keep track of up-to-date score upper bounds. Instead, the updates to these upper bounds can be deferred to a later step, or can be reduced to a much more compact set of *necessary* updates for more efficient computation. An NRA variant, called *LARA*, has been introduced based on a lattice structure that keeps a leader object for each subset of the ranked inputs. These leader objects provide score upper bounds for objects that have not been seen yet on their corresponding inputs. The top-k processing algorithm proceeds in two successive phases:

—*A growing phase*. Ranked inputs are sequentially scanned to compose a candidate set. The seen objects in different inputs are added to the candidate set. A set W_k, containing the k objects with highest score lower bounds, is remembered at each

step. The candidate set construction is complete when the threshold value (the score upper bound of any unseen object) is below the minimum score of W_k. At this point, we are sure that the top-k query answer belongs to the candidate set.

—*A shrinking phase*. Materialized top-k candidates are pruned gradually, by computing their score upper bounds, until the final top-k answer is obtained.

Score upper bound computation makes use of the lattice to minimize the number of required accesses to the ranked inputs by eliminating the need to access some inputs once they become useless for future operations. Different adaptation of LARA in various settings have been proposed including providing answers online or incrementally, processing rank join queries, and working with different rank aggregation functions.

Another example of *no random access* top-k algorithms is the J^* algorithm [Natsev et al. 2001]. The J^* algorithm adopts a top-k join query model (Section 2.1), where the top-k join results are computed by joining multiple ranked inputs based on a join condition, and scoring the outcome join results based on a monotone score aggregation function. The J^* algorithm is based on the A^* search algorithm. The idea is to maintain a priority queue of partial and complete join combinations, ordered on the upper bounds of their total scores. At each step, the algorithm tries to complete the join combination at queue top by selecting the next input stream to join with the partial join result, and retrieving the next object from that stream. The algorithm reports the next top join result as soon as the join result at queue top includes an object from each ranked input.

For each input stream, a *variable* is defined whose possible assignments are the set of stream objects. A *state* is defined as a set of variable assignments, and a state is *complete* if it instantiates all variables. The problem of finding a valid join combination with maximum score reduces to finding an assignment for all the variables, based on join condition, that maximizes the overall score. The score of a state is computed by exploiting the monotonicity of the score aggregation function. That is, the scores of complete states are computed by aggregating the scores of their instantiated variables, while the scores of incomplete states are computed by aggregating the scores of their instantiated variables, and the score upper bounds of their noninstantiated variables. The score upper bounds of noninstantiated variables are equal to the last seen scores in the corresponding ranked inputs.

3.3. Sorted Access with Controlled Random Probes

Top-k processing methods in this category assume that at least one source provides sorted access, while random accesses are scheduled to be performed only when needed. The Upper and Pick algorithms [Bruno et al. 2002b; Marian et al. 2004] are examples of these methods.

The Upper and Pick algorithms are proposed in the context of Web-accessible sources. Such sources usually have large variation in the allowed access methods, and their costs. Upper and Pick assume that each source can provide a sorted and/or random access to its ranked input, and that at least one source supports sorted access. The main purpose of having at least one sorted-access source is to obtain an initial set of candidate objects. Random accesses are controlled by selecting the best candidates, based on score upper bounds, to complete their scores.

Three different types of sources are defined based on the supported access method: (1) *S-Source* that provides sorted access, (2) *R-Source* that provides random access, and (3) *SR-Source* that provides both access methods. The initial candidate set is obtained using at least one S-Source. Other R-Sources are probed to get the required partial scores as required.

Algorithm 3. Upper [Bruno et al. 2002b]

```
1: Define Candidates as priority queue based on F̄(.)
2: T = 1            {Score upper bound for all unseen tuples}
3: returned=0
4: while returned < k do
5:         if Candidates ≠ φ then
6:                 select from Candidates the object t_top with the maximum F̄(.)
7:         else
8:                 t_top is undefined
9:       end if
10:        if t_top is undefined or F̄(t_top) < T then
11:            Use a round-robin policy to choose the next sorted list L_i to access.
12:            t = L_i.getNext()
13:            if t is new object then
14:                        Add t to Candidates
15:            else
16:                        Update F̄(t), and update Candidates accordingly
17:         end if
18:            T = F(p̄_1, ..., p̄_m)
19:       else if F(t_top) is completely known then
20:            Report (t_top, F(t_top))
21:            Remove t_top from Candidates
22:            returned = returned +1
23:       else
24:            L_i = SelectBestSource(t_top)
25:            Update F̄(t_top) with the value of predicate p_i via random probe to L_i
26:    end if
27: end while
```

The Upper algorithm, as illustrated by Algorithm 3, probes objects that have considerable chances to be among the top-k objects. In Algorithm 3, it is assumed that objects' scores are normalized in the range [0, 1]. Candidate objects are retrieved first from sorted sources, and inserted into a priority queue based on their score upper bounds. The upper bound of unseen objects is updated when new objects are retrieved from sorted sources. An object is reported and removed from the queue when its exact score is higher than the score upper bound of unseen objects. The algorithm repeatedly selects the best source to probe next to obtain additional information for candidate objects. The selection is performed by the *SelectBestSource* function. This function could have several implementations. For example, the source to be probed next can be the one which contributes the most in decreasing the uncertainty about the top candidates.

In the Pick algorithm, the next object to be probed is selected so that it minimizes a distance function, which is defined as the sum of the differences between the upper and lower bounds of all objects. The source to be probed next is selected at random from all sources that need to be probed to complete the score of the selected object.

A related issue to controlling the number of random accesses is the potentially expensive evaluation of ranking predicates. The full evaluation of user-defined ranking predicates is not always tolerable. One reason is that user-defined ranking predicates are usually defined only at at query time, limiting the chances of benefiting from precomputations. Another reason is that ranking predicates might access external autonomous

Table II. Object Scores Based on Different Ranking Predicates

Object	p_1	p_2	p_3	$F = p_1 + p_2 + p_3$
a	0.9	1.0	0.5	2.4
b	0.6	0.4	0.4	1.4
c	0.4	0.7	0.9	2.0
d	0.3	0.3	0.5	1.1
e	0.2	0.4	0.2	0.8

Table III. Finding the Top-2 Objects in MPro

Step	Action	Priority queue	Output
1	Initialize	$[a : 2.9, b : 2.6, c : 2.4, d : 2.3, e : 2.2]$	{}
2	After *probe*(a, p_2)	$[a : 2.9, b : 2.6, c : 2.4, d : 2.3, e : 2.2]$	{}
3	After *probe*(a, p_3)	$[b : 2.6, a : 2.4, c : 2.4, d : 2.3, e : 2.2]$	{}
4	After *probe*(b, p_2)	$[c : 2.4, d : 2.3, e : 2.2, b : 2.0]$	{a:2.4}
5	After *probe*(c, p_2)	$[d : 2.3, e : 2.2, c : 2.1, b : 2.0]$	{a:2.4}
6	After *probe*(d, p_2)	$[e : 2.2, c : 2.1, b : 2.0, d : 1.6]$	{a:2.4}
7	After *probe*(e, p_2)	$[c : 2.1, b : 2.0, e : 1.6, d : 1.6]$	{a:2.4}
8	After *probe*(c, p_3)	$[b : 2.0, e : 1.6, d : 1.6]$	{a:2.4,c:2.0}

sources. In these settings, optimizing the number of times the ranking predicates are invoked is necessary for efficient query processing.

These observations motivated the work of Chang and Hwang [2002] and Hwang and Chang [2007b] which introduced the concept of "necessary probes," to indicate whether a predicate probe is absolutely required or not. The proposed *Minimal Probing (MPro)* algorithm adopts this concept to minimize the predicate evaluation cost. The authors considered a top-k query with a scoring function F defined on a set of predicates $p_1 \cdots p_n$. The score upper bound of an object t, denoted $\overline{F}(t)$, is computed by aggregating the scores produced by each of the evaluated predicates and assuming the maximum possible score for unevaluated predicates. The aggregation is done using a monotonic function, for example, weighted summation. Let *probe*(t, p) denote probing predicate p of object t. It has been shown that *probe*(t, p) is necessary if t is among the current top-k objects based on the score upper bounds. This observation has been exploited to construct *probing schedules* as sequences of necessary predicate probes.

The MPro algorithm works in two phases. First, in the initialization phase, a priority queue is initialized based on the score upper bounds of different objects. The algorithm assumes the existence of at least one *cheap* predicate where sorted access is available. The initial score upper bound of each object is computed by aggregating the scores of the *cheap* predicates and the maximum scores of expensive predicates. Second, in the probing phase, the object at queue top is removed, its next unevaluated predicate is probed, its score upper bound is updated, and the object is reinserted back to the queue. If the score of the object at queue top is already complete, the object is among the top-k answers and it is moved to the output queue. Finding the optimal probing schedule for each object is shown to be in NP-Hard complexity class. Optimal probing schedules are thus approximated using a greedy heuristic defined using the benefit and cost of each predicate, which are computed by sampling the ranked lists.

We illustrate the processing of *MPro* using the next example. Table II shows the scores of one cheap predicate, p_1, and two expensive predicates, p_2 and p_3, for a list of objects $\{a, b, c, d, e\}$. Table III illustrates how MPro operates, based a scoring function $F = p_1 + p_2 + p_3$, to find the top-2 objects. We use the notation *object:score* to refer to the score upper bound of the object.

The goal of the MPro algorithm is to minimize the cost of random access, while assuming the cost of sorted access is cheap. A generalized algorithm, named *NC*, has

been introduced in Hwang and Chang [2007a] to include the cost of sorted access while scheduling predicates probing. The algorithm maintains the current top-k objects based on their scores upper bounds. At each step, the algorithm identifies the set of necessary probing alternatives to determine the top-k objects. These alternatives are probing the unknown predicates for the current top-k objects using either sorted or random access. The authors proved that it is sufficient to consider the family of algorithms that perform sorted access before random access in order to achieve the optimal scheduling. Additionally, to provide practical scheduling of alternatives, the NC algorithm restricts the space of considered schedules to those that only perform sorted accesses up to certain depth Δ, and follow a fixed schedule $\mathcal{H}$ for random accesses. The algorithm first attempts to perform a sorted access. If there is no sorted access among the probing alternatives, or all sorted accesses are beyond the depth Δ, a random access takes place based on the schedule $\mathcal{H}$. The parameters Δ and $\mathcal{H}$ are estimated using sampling.

4. IMPLEMENTATION LEVEL

In this section, we discuss top-k processing methods based on the design choices they make regarding integration with database systems. Some techniques are designed as application-level solutions that work outside the database engine, while others involve low level modifications to the query engine. We describe techniques belonging to these two categories in the next sections.

4.1. Application Level

Top-k query processing techniques that are implemented at the application level, or middleware, provide a ranked retrieval of database objects, without major modification to the underlying database system, particularly the query engine. We classify application level top-k techniques into Filter-Restart methods, and Indexes/Materialized Views methods.

4.1.1. Filter-Restart. Filter-Restart techniques formulate top-k queries as *range selection queries* to limit the number of retrieved objects. That is, a top-k query that ranks objects based on a scoring function F, defined on a set of scoring predicates $p_1, \ldots, p_m$, is formulated as a range query of the form "find objects with $p_1 > T_1 \; and \; \cdots \; and \; p_m > T_m$", where T_i is an estimated cutoff threshold for predicate p_i. Using a range query aims at limiting the retrieved set of objects to the necessary objects to answer the top-k query. The retrieved objects have to be ranked based on F to find the top-k answers.

Incorrect estimation of cutoff threshold yields one of two possibilities: (1) if the cutoff is overestimated, the retrieved objects may not be sufficient to answer the top-k query and the range query has to be restarted with looser thresholds, or (2) if the cutoff is under-estimated, the number of retrieved objects will be more than necessary to answer the top-k query. In both cases, the performance of query processing degrades.

One proposed method to estimate the cutoff threshold is using the available statistics such as histograms [Bruno et al. 2002a], where the scoring function is taken as the distance between database objects and a given query point q. Multidimensional histograms on objects' attributes (dimensions) are used to identify the cutoff distance from q to the potential top-k set. Two extreme strategies can be used to select such cutoff distance. The first strategy, named the *restarts strategy*, is to select the search distance as tight as possible to just enclose the potential top-k objects. Such a strategy might retrieve less objects than the required number (k), necessitating restarting the search with a larger distance. The second strategy, named the *no-restarts strategy*, is to

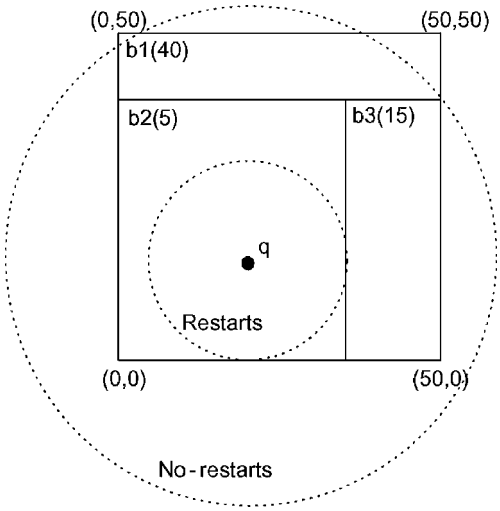

Fig. 7. An example of restarts and no-restarts strategies in the Filter-Restart approach [Bruno et al. 2002a].

choose the search distance large enough to include all potential top-k objects. However, this strategy may end up retrieving a large number of unnecessary objects.

We illustrate the two strategies using Figure 7, where it is required to find the 10 closest objects to q. The rectangular cells are two-dimensional histogram bins annotated with the number of data objects in each bin. The inner circle represents the restarts strategy where, hopefully, exactly 10 objects will be retrieved: five objects from bin $b3$, and five objects from bin $b2$. This strategy will result in restarts if less than 10 objects are retrieved. On the other hand, the no-restarts strategy uses the outer circle, which completely encloses bins $b2$ and $b3$, and thus ensures that at least 20 objects will be retrieved. However, this strategy will retrieve unnecessary objects.

To find the optimal search distance, query workload is used as a training set to determine the number of returned objects for different search distances and q locations. The optimal search distance is approximated using an optimization algorithm running over all the queries in the workload. The outcome of the optimization algorithm is a search distance that is expected to minimize the overall number of retrieved objects, and the number of restarts.

A probabilistic approach to estimate cutoff threshold was proposed by Donjerkovic and Ramakrishnan [1999]. A top-k query based on an attribute X is mapped into a selection predicate $\sigma_{X>T}$, where T is the estimated cutoff threshold. A probabilistic model is used to search for the selection predicate that would minimize the overall expected cost of restarts. This is done by constructing a probability distribution over the cardinalities of possible selection predicates in the form of *(cardinality-value, probability)* pairs, where the cardinality-value represents the number of database tuples satisfying the predicate, and the probability represents the degree of certainty in the correctness of the cardinality-value, which reflects the potentially imperfect statistics on the underlying data.

The probability distribution is represented as a vector of equi-probable cardinality points to avoid materializing the whole space. Every cardinality point is associated with a cost estimate representing the initial query processing cost, and the cost of possible restart. The goal is to find a query plan that minimizes the expected cost over all cardinality values. To be consistent with the existing cardinality estimates, the cardinality distribution has an average equal to the average cardinality obtained from

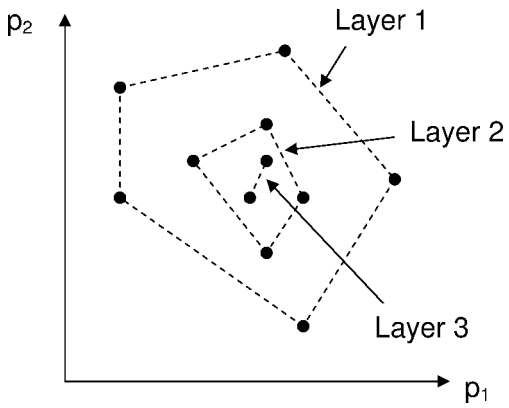

Fig. 8. Convex hulls in two-dimensional space.

existing histograms. The maintained cardinality values are selected at equi-distant steps to the left and right of the average cardinality, with a predetermined total number of points. The stepping distance, to the left and right of average point, is estimated based on the worst case estimation error of the histogram.

4.1.2. Using Indexes and Materialized Views. Another group of *application level* top-k processing techniques use specialized indexes and materialized views to improve the query response time at the expense of additional storage space. Top-k indexes usually make use of special geometric properties of the scoring function to index the underlying data objects. On the other hand, materialized views maintain information that is expensive to gather online, for example, a sorting for the underlying objects based on some scoring function, to help compute top-k queries efficiently.

4.1.2.1. Specialized Top-k Indexes. One example of specialized top-k indexes is the *Onion Indices* [Chang et al. 2000]. Assume that tuples are represented as n–dimensional points where each dimension represents the value of one scoring predicate. The *convex hull* of these points is defined as the boundary of the smallest convex region that encloses them. The geometrical properties of the convex hull guarantee that it includes the top-1 object (assuming a linear scoring function defined on the dimensions). Onion Indices extend this observation by constructing layered convex hulls, shown in Figure 8, to index the underlying objects for efficient top-k processing.

The Onion Indices return the top-1 object by searching the points of the outmost convex hull. The next result (the top-2 object) is found by searching the remaining points of the outmost convex hull, and the points belonging to the next layer. This procedure continues until all of the top-k results are found. Although this indexing scheme provides performance gain, it becomes inefficient when the top-k query involves additional constraints on the required data, such as range predicates on attribute values. This is because the convex hull structure will be different for each set of constraints. A proposed work-around is to divide data objects into smaller clusters, index them, and merge these clusters into larger ones progressively. The result is a hierarchical structure of clusters, each of which has its own Onion Indices. A constrained query can probably be answered by indices of smaller clusters in the hierarchy. The construction of Onion Indices has an asymptotic complexity of $O(n^{d/2})$, where d is the number of dimensions and n is the number of data objects.

The idea of multilayer indexing has been also adopted by Xin et al. [2006] to provide *robust* indexing for top-k queries. Robustness is defined in terms of providing the best

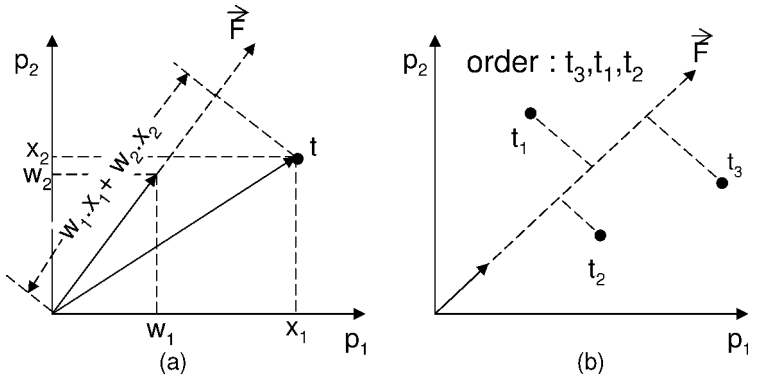

Fig. 9. Geometric representation of tuples and scoring function: (a) projection of tuple $t = (x_1, x_2)$ on scoring function vector (w_1, w_2); (b) order based on obtained scores.

possible performance in worst case scenario, which is fully scanning the first k layers to find the top-k answers. The main idea is that if each object o_i is pushed to the deepest possible layer, its retrieval can be avoided if it is unnecessary. This is accomplished by searching for the minimum rank of each object o_i in all linear scoring functions. Such rank represents the layer number, denoted $l^*(o_i)$, where object o_i is pushed to. For n objects having d scoring predicates, computing the exact layer numbers for all objects has a complexity of $O(n^d \log n)$, which is an overkill when n or d are large. Approximation is used to reduce the computation cost. An approximate layer number, denoted $l(o_i)$, is computed such that $l(o_i) \leq l^*(o_i)$, which ensures that no false positives are produced in the top-k query answer. The complexity of the approximation algorithm is $O(2^d n(\log n)^{r(d)-1})$, where $r(d) = \lceil \frac{d}{2} \rceil + \lfloor \frac{d}{2} \rfloor \lceil \frac{d}{2} \rceil$.

Ranked Join Indices [Tsaparas et al. 2003] is another top-k index structure, based on the observation that the projection of a vector representing a tuple t on the normalized scoring function vector $\overrightarrow{F}$ reveals t's rank based on F. This observation applies to any scoring function that is defined as a linear combination of the scoring predicates. For example, Figure 9 shows a scoring function $F = w_1.p_1 + w_2.p_2$, where p_1 and p_2 are scoring predicates, and w_1 and w_2 are their corresponding weights. In this case, we have $\overrightarrow{F} = (w_1, w_2)$. Without loss of generality, assume that $\| \overrightarrow{F} \| = 1$. We can obtain the score of $t = (x_1, x_2)$ by computing the length of its projection on $\overrightarrow{F}$, which is equivalent to the dot product $(w_1, w_2) \odot (x_1, x_2) = w_1.x_1 + w_2.x_2$. By changing the values of w_1 and w_2, we can sweep the space using a vector of increasing angle to represent any possible linear scoring function. The tuple scores given by an arbitrary linear scoring function can thus be materialized.

Before materialization, tuples that are dominated by more than k tuples are discarded because they do not belong to the top-k query answer of any linear scoring function. The remaining tuples, denoted as the *dominating set* $\mathcal{D}_k$, include all possible top-k answers for any possible linear scoring function. Algorithm 4 describes how to construct the dominating set $\mathcal{D}_k$ with respect to a scoring function that is defined as a linear combination of predicates p_1 and p_2. The algorithm starts by first sorting all the tuples based on p_1, and then scanning the sorted tuples. A priority queue Q is maintained to keep the top-k tuples, encountered so far, based on predicate p_2. The first k tuples are directly copied to $\mathcal{D}_k$, while subsequent tuples are examined against the minimum value of p_2 in Q. If the p_2 value of some tuple t is less than the minimum p_2 value in Q, then t is discarded, since there are at least k objects with greater p_1 and p_2 values.

Algorithm 4. Ranked Join Indices: GetDominatingSet [Tsaparas et al. 2003]

```
Define Q as a priority queue based on p2 values
Define D_k as the dominating set. Initially, set D_k = φ
Sort tuples in non-increasing order of p1 values.
for each tuple t_i do
    if |Q| < k then
        D_k = D_k ∪ t_i
        insert (t_i, p2(t_i)) in Q
    else if p2(t_i) ≤ (minimum p2 value in Q) then
        skip t_i
    else
        D_k = D_k ∪ t_i
        insert (t_i, p2(t_i)) in Q
        if |Q| > k then
            delete the minimum element of Q
        end if
    end if
end for
Return D_k
```

Algorithm 5. Ranked Join Indices: ConstructRJI ($\mathcal{D}_k$) [Tsaparas et al. 2003]

```
Require: D_k: The dominating tuple set
V = φ        { the separating vector set}
for each t_i, t_j ∈ D_k, t_i ≠ t_j do
    e_sij = separating vector for (t_i, t_j)
    insert (t_i, t_j) and their corresponding separating vector e_sij in V
end for
Sort V in non-decreasing order of vector angles a(e_sij)
A_k = top-k tuples in D_k with respect to predicate p1
for each (t_i, t_j) ∈ V do
    if both t_i, t_j ∈ A_k ∨ both t_i, t_j ∉ A_k then
        No change in A_k by e_sij
    else if t_i ∈ A_k ∧ t_j ∉ A_k then
        Store (a(e_sij), A_k) into index
        Replace t_i with t_j in A_k
    else if t_i ∉ A_k ∧ t_j ∈ A_k then
        Store (a(e_sij), A_k) into index
        Replace t_j with t_i in A_k
    end if
end for
Store (a(p2→), A_k)
```

We now describe how Ranked Join Indices materialize top-k answers for different scoring functions. Figure 10 shows how the order of tuples change based on the scoring function vector. For two tuples t_1 and t_2, there are two possible cases regarding their relative positions:

Case 1. The line connecting the two tuples has a positive slope. In this case, their relative ranks are the same for any scoring function e. This case is illustrated in Figure 10(a).

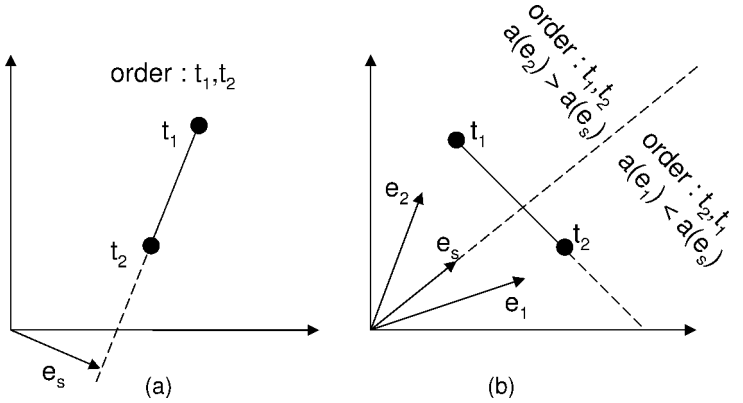

Fig. 10. Possible relative positions of tuples t_1 and t_2: (a) positive slope of the line connecting $t1$ and $t2$; (b) negative slope of the line connecting $t1$ and $t2$.

Case 2. The line connecting the two tuples has a negative slope. In this case, there is a vector that separates the space into two subspaces, where the tuples' order in one of them is the inverse of the other. This vector, denoted as e_s, is perpendicular to the line connecting the two tuples t_1 and t_2. This case is illustrated in Figure 10(b).

Based on the above observation, we can index the order of tuples by keeping a list of all scoring functions vectors (along with their angles) that switch tuples' order. Algorithm 5 describes the details of constructing Ranked Join Indices. The algorithm initially finds the separating vectors between each pair of tuples and sorts these vectors based on their angles. Then, it ranks tuples based on some scoring function (e.g., $F = p_1$) and starts scanning the separating vectors in order. Whenever a vector $e_{s_{ij}}$ is found such that it changes the order of tuples t_i and t_j (case 2), the vector's angle and its corresponding top-k set are stored in a B-tree index, using angle as the index key. The index construction algorithm has a time complexity of $O(|\mathcal{D}_k|^2 log\, |\mathcal{D}_k|)$ and a space complexity of $O(|\mathcal{D}_k|k^2)$.

At query time, the vector corresponding to the scoring function, specified in the query, is determined, and its angle is used to search the B-tree index for the corresponding top-k set. The exact ranking of tuples in the retrieved top-k set is computed, and returned to the user. The query processing algorithm has a complexity of $O(log|\mathcal{D}_k| + k\ log\ k)$.

4.1.2.2. Top-k Materialized Views. Materialized views have been studied in the context of top-k processing as a means to provide efficient access to scoring and ordering information that is expensive to gather during query execution. Using materialized views for top-k processing has been studied in the PREFER system [Hristidis et al. 2001; Hristidis and Papakonstantinou 2004], which answers preference queries using materialized views. Preference queries are represented as ORDER BY queries that return sorted answers based on predefined scoring predicates. The user preference of a certain tuple is captured by an arbitrary weighted summation of the scoring predicates. The objective is to answer such preference queries using a reasonable number of materialized views.

The proposed method keeps a number of materialized views based on different weight assignments of the scoring predicates. Specifically, each view v ranks the entire set of underlying tuples based on a scoring function F_v defined as a weighted summation of the scoring predicates using some weight vector $\overrightarrow{v}$. For a top-k query with an arbitrary weight vector $\overrightarrow{q}$, the materialized view that best matches $\overrightarrow{q}$ is selected to find query answer. Such view is found by computing a position marker for each view to determine

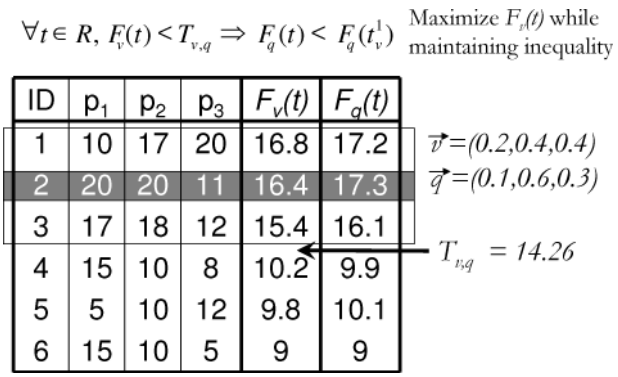

Fig. 11. Finding top-1 object based on some materialized view.

the number of tuples that need to be fetched from that view to find query answer. The best view is the one with the least number of fetched tuples.

Top-k query answers in the PREFER system are pipelined. Let n be the number of tuples fetched from view v after computing v's position marker. If $n \geq k$, then processing terminates. Otherwise, the n tuples are reported, and a new position marker is computed for v. The process is repeated until k tuples are reported. Computing position markers follows the next procedure. Assume a top-k query q is executed against a relation R. The first marker position for view v is the maximum value $T^1_{v,q}$ with the following property $\forall t \in R : F_v(t) < T^1_{v,q} \Rightarrow F_q(t) < F_q(t^1_v)$, where t^1_v is the top tuple in v. At next iterations t^{top}_v, the unreported v tuple with the highest score in v, replaces t^1_v in computing the marker position of v.

We illustrate how PREFER works using Figure 11, which depicts the procedure followed to find the top-1 object. The depicted view materializes a sorting of the relation R based on weighted summation of the scoring predicates p_1, p_2, and p_3 using the weight vector $\vec{v}$. However, we are interested in the top-1 object based on another weight vector $\vec{q}$. An optimization problem is solved over the set of materialized views to find the view with the shortest prefix that needs to be fetched to answer this top-1 query. The value $T_{v,q}$ is computed for each view v, such that every tuple in v with $F_v(t) < T_{v,q}$ cannot be the top-1 answer (i.e., there exists another tuple with higher $F_q(.)$ value). This can be verified using the first tuple in the view t^1_v. Once $T_{v,q}$ is determined, the prefix from v above $T_{v,q}$ is fetched and sorted based on F_q to find the top-1 object. For example, in Figure 11, a prefix of length 3 needs to be fetched from the depicted view to find out the top-1 tuple based on F_q. Among the retrieved three tuples, the second tuple is the required top-1 answer. Finding the top-i tuple operates similarly, in a pipelined fashion, by looking for a new $T_{v,q}$ value for each i value, such that there exist at least i tuples with larger $F_q(.)$ values than any other tuple below $T_{v,q}$.

Using materialized views for top-k processing, with linear scoring functions, has been also studied in the LPTA technique proposed by Das et al. [2006]. Top-k answers are obtained by sequentially reading from materialized views, built using the answers of previous queries, and probing the scoring predicates of the retrieved tuples to compute their total scores. The main idea is to choose an optimal subset among all available views to minimize the number of accessed tuples. For example, in the case of linear scoring functions defined on two scoring predicates, only the views with the closest vectors to the query vector in anticlockwise and clockwise directions need to be considered to efficiently answer the query. For example in Figure 12, only the views whose vectors are v_2 and v_3 are considered to compute the query Q. The authors showed that selecting

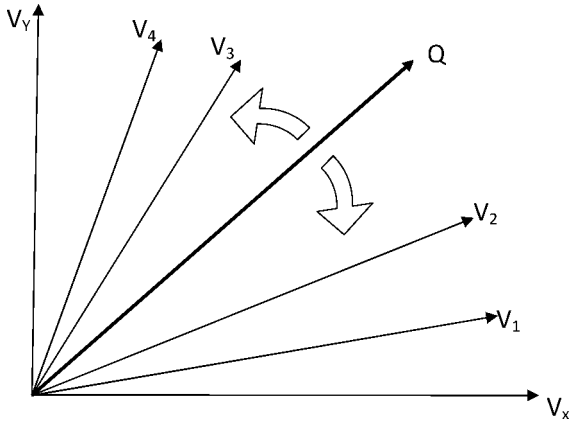

Fig. 12. Choosing the optimal subset of materialized views in the LPTA algorithm [Das et al. 2006].

further views in this case is suboptimal. The LPTA algorithm finds the top-k answers by scanning both views, and computing the scores of the retrieved tuples while maintaining a set of top-k candidates. The stoping criterion is similar to TA; the algorithm terminates when the minimum score in the candidate set has a score greater than the maximum score of the unseen tuples, denoted T. The value of T is computed using linear programming. Specifically, each view provides a linear constraint that bounds the space where the non-retrieved tuples reside. Constraints of different views form a convex region, and the maximum score of the unseen tuples is obtained by searching this region.

To answer top-k queries in higher dimensions, the authors proved that it is sufficient to use a subset of the available views with size less than or equal to the number of dimensions. An approximate method is used to determine which subset of views is selected based on estimates of the execution cost for each subset. To estimate the cost of a specific subset of views, a histogram of tuples' scores is built using the available histograms of scoring predicates. A greedy algorithm is used to determine the optimal subset of views by incrementally adding the view that provides the minimum estimated cost.

4.2. Engine Level

The main theme of the techniques discussed in this section is their tight coupling with the query engine. This tight coupling has been realized through multiple approaches. Some approaches focus on the design of efficient specialized rank-aware query operators. Other approaches introduce an algebra to formalize the interaction between ranking and other relational operations (e.g., joins and selections). A third category addresses modifying query optimizers, for example, changing optimizers' plan enumeration and cost estimation procedures, to recognize the ranking requirements of top-k queries. Treating ranking as a first-class citizen in the query engine provides significant potential for efficient execution of top-k queries. We discuss the techniques that belong to the above categories in the next sections.

4.2.1. Query Operators. The techniques presented in this section provide solutions that embed rank-awareness within query operators. One important property that is satisfied by many of these operators is *pipelining*. Pipelining allows for reporting query answers without processing all of the underlying data if possible, and thus minimizing query response time. In pipelining, the next object produced by one query operator is fed into a subsequent operator upon request. Generally, algorithms that require random access are unsuitable for pipelining. The reason is that requesting objects by their

identifiers breaks the pipeline by materializing specific objects, which could incur a large overhead. TA and CA (discussed in Section 3.1) are thus generally unsuitable for pipelining. Although the NRA algorithm (discussed in Section 3.2) does not require random access, it is not also capable of pipelining since the reported objects do not have associated exact scores. Hence, the output of one NRA process cannot serve as a valid input to another NRA process.

One example of rank-aware query operators that support pipelining is the Rank-Join operator [Ilyas et al. 2004a], which integrates the joining and ranking tasks in one efficient operator. Algorithm 6 describes the main Rank-Join procedure. The algorithm has common properties with the NRA algorithm [Fagin et al. 2001] (described in Section 3.2). Both algorithms perform sorted access to get tuples from each data source. The main difference is that the NRA algorithm assumes that each partially seen tuple has a valid score that can be completely computed if the values of the currently unknown tuple's scoring predicates are obtained. This assumption cannot be made for the case of joining tuples from multiple sources, since arbitrary subsets of the Cartesian product of tuples may end up in the join result based on the join condition. For this reason, the Rank-Join algorithm maintains the scores of the completely seen join combinations only. As a result, the Rank-Join algorithm reports the exact scores of the top-k tuples, while the NRA algorithm reports bounds on tuples' scores. Another difference is that the NRA algorithm has strict access pattern that requires retrieval of a new tuple from each source at each iteration. The Rank-Join algorithm does not impose any constraints on tuples retrieval, leaving the access pattern to be specified by the underlying join algorithm.

Similarly to NRA algorithm, the Rank-Join algorithm scans input lists (the joined relations) in the order of their scoring predicates. Join results are discovered incrementally as the algorithm moves down the ranked input relations. For each join result j, the algorithm computes a score for j using a score aggregation function F, following the top-k join query model (Section 2.1). The algorithm maintains a threshold T bounding the scores of join results that are not discovered yet. The top-k join results are obtained when the minimum score of the k join results with the maximum $F(.)$ values is not below the threshold T.

A two-way hash join implementation of the Rank-Join algorithm, called *Hash Rank Join Operator (HRJN)*, was introduced in Ilyas et al. [2004a]. HRJN is based on symmetrical hash join. The operator maintains a hash table for each relation involved in

Algorithm 6. Rank Join [Ilyas et al. 2004a]

—Retrieve tuples from input relations in descending order of their individual scores p_i's. For each new retrieved tuple t:

(1) Generate new valid join combinations between t and seen tuples in other relations.
(2) For each resulting join combination j, compute $F(j)$.
(3) Let $p_i^{(max)}$ be the top score in relation i, that is, the score of the first tuple retrieved from relation i. Let $\overline{p}_i$ be the last seen score in relation i. Let T be the maximum of the following m values:
$F(\overline{p}_1, p_2^{max}, \ldots, p_m^{max})$,
$F(p_1^{max}, \overline{p}_2, \ldots, p_m^{max})$,
...
$F(p_1^{max}, p_2^{max}, \ldots, \overline{p}_m)$.
(4) Let A_k be a set of k join results with the maximum $F(.)$ values, and M_k be the lowest score in A_k. Halt when $M_k \geq T$.

—Report the join results in A_k ordered on their $F(.)$ values.

the join process, and a priority queue to buffer the join results in the order of their scores. The hash tables hold input tuples seen so far and are used to compute the valid join results. The HRJN operator implements the traditional iterator interface of query operators. The details of the *Open* and *GetNext* methods are given by Algorithms 7 and 8, respectively. The *Open* method is responsible for initializing the necessary data structure; the priority queue Q, and the left and right hash tables. It also sets T, the score upper bound of unseen join results, to the maximum possible value.

Algorithm 7. HRJN: Open(L_1, L_2) [Ilyas et al. 2004a]

Require: L_1: Left ranked input, L_2: Right ranked input
Create a priority queue Q to order join results based on $F(.)$ values
Build two hash tables for L_1 and L_2
Set threshold T to the maximum possible value of F
Initialize $\overline{p}_1$ and p_1^{max} with the maximum score in L_1
Initialize $\overline{p}_2$ and p_2^{max} with the maximum score in L_2
L_1.Open()
L_2.Open()

Algorithm 8. HRJN: GetNext [Ilyas et al. 2004a]

```
if |Q| > 0 then
        j_top = peek at top element in Q
        if F(j_top) ≥ T then
                Remove j_top from Q
                Return j_top
    end if
end if
loop
         Determine next input to access, L_i
        t = L_i.GetNext()
        if t is the first seen tuple in L_i then
            p_i^max = p_i(t)
    end if
        p̄_i = p_i(t)
        T = MAX(F(p_1^max, p̄_2), F(p̄_1, p_2^max))
        insert t in L_i Hash table
        probe the other hash table using t's join key
        for all valid join combination j do
                Compute F(j)
                Insert j in Q
    end for
         if |Q| > 0 then
                j_top = peek at top element in Q
                if F(j_top) ≥ T then
                                break loop
           end if
    end if
end loop
Remove top tuple j_top from Q
Return j_top
```

The *GetNext* method remembers the two top scores, p_1^{max} and p_2^{max}, and the last seen scores, $\overline{p}_1$ and $\overline{p}_2$ of its left and right inputs. Notice that $\overline{p}_1$ and $\overline{p}_2$ are continuously updated as new tuples are retrieved from the input relations. At any time during query execution, the threshold T is computed as the maximum of $F(p_1^{max}, \overline{p}_2)$ and $F(\overline{p}_1, p_2^{max})$. At each step, the algorithm reads tuples from either the left or right inputs, and probes the hash table of the other input to generate join results. The algorithm decides which input to poll at each step, which gives flexibility to optimize the operator for fast generation of join results based on the joined data. A simplistic strategy is accessing the inputs in a round-robin fashion. A join result is reported if its score is not below the scores of all discovered join results, and the threshold T.

Other examples of top-k operators that are suitable for pipelining are the NRA-RJ operator [Ilyas et al. 2002], and the J^* algorithm [Natsev et al. 2001]. The NRA-RJ operator extends the NRA algorithm [Fagin et al. 2001] using an efficient query operator that can serve valid input to other NRA-RJ operators in the query pipeline. The J^* algorithm [Natsev et al. 2001] (discussed in Section 3.2) supports pipelining since it does not require random access to its inputs, and produces join results with complete scores.

Li et al. [2006] introduced rank-aware query operators that work under the top-k aggregate query model (Section 2.1). Top-k aggregate queries report the k groups (based on some grouping columns) with the highest aggregate values (e.g., sum). The conventional processing of such queries follows a materialize-group-sort scheme, which can be inefficient if only the top-k groups are required. Moreover, it is common, in this kind of queries, to use ad hoc aggregate functions that are specified only at query time for data exploration purposes. Supporting such ad hoc aggregate functions is challenging since they cannot benefit from any existing precomputations.

Two fundamental principles have been proposed in Li et al. [2006] to address the above challenges. The first principle, *Group-Ranking*, dictates the order in which groups are probed during top-k processing. The authors proposed prioritizing group access by incrementally consuming tuples from the groups with the maximum possible aggregate values. This means that it might not be necessary to complete the evaluation of some groups not included in the current top-k. Knowing the maximum possible aggregate values beforehand is possible if information regarding the cardinality of each group can be obtained. This information is typically available in environments such as OLAP, where aggregation and top-k queries are dominant.

The second principle, *Tuple-Ranking*, dictates the order in which tuples should be accessed from each group. In aggregate queries, each tuple has a scoring attribute, usually referred to as the *measure* attribute, which contributes to the aggregate score of tuple's group, for example, the *salary* attribute for the aggregate function *sum*(*salary*). The authors showed that the tuple order that results in the minimum *tuple depth* (the number of accessed tuples from each group), is among three tuple orders, out of all possible permutations: *Descending Tuple Score Order*, *Ascending Tuple Score Order*, and *Hybrid Order*, which chooses the tuple with either the highest or lowest score among unseen tuples.

The two above principles were encapsulated in a query operator, called *rankaggr*. The new operator eagerly probes the groups instead of waiting for all groups to be materialized by the underlying query subtree. The next group to probe is determined according to the maximum possible scores of all valid groups, and the next tuple is drawn from this group. As a heuristic, tuples are accessed from any group in descending score order. When a group is exhausted, its aggregate value is reported. This guarantees pipelining the resulting groups in the correct order with respect to their aggregate values.

Rank: μ, with a ranking predicate p • $t \in \mu_p(R_{\mathcal{P}})$ iff $t \in R_{\mathcal{P}}$ • $t_1 <_{\mu_p(R_{\mathcal{P}})} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t_2]$
Selection: σ, with a Boolean condition c • $t \in \sigma_c(R_{\mathcal{P}})$ iff $t \in R_{\mathcal{P}}$ and t satisfies c • $t_1 <_{\sigma_c(R_{\mathcal{P}})} t_2$ iff $t_1 <_{R_{\mathcal{P}}} t_2, i.e., \overline{\mathcal{F}}_{\mathcal{P}}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}}[t_2]$
Union: $\cup$ • $t \in R_{\mathcal{P}_1} \cup S_{\mathcal{P}_2}$ iff $t \in R_{\mathcal{P}_1}$ or $t \in S_{\mathcal{P}_2}$ • $t_1 <_{R_{\mathcal{P}_1}\cup S_{\mathcal{P}_2}} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_2]$
Intersection: $\cap$ • $t \in R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2}$ iff $t \in R_{\mathcal{P}_1}$ and $t \in S_{\mathcal{P}_2}$ • $t_1 <_{R_{\mathcal{P}_1}\cap S_{\mathcal{P}_2}} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_2]$
Difference: $-$ • $t \in R_{\mathcal{P}_1} - S_{\mathcal{P}_2}$ iff $t \in R_{\mathcal{P}_1}$ and $t \notin S_{\mathcal{P}_2}$ • $t_1 <_{R_{\mathcal{P}_1}-S_{\mathcal{P}_2}} t_2$ iff $t_1 <_{R_{\mathcal{P}_1}} t_2, i.e., \overline{\mathcal{F}}_{\mathcal{P}_1}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}_1}[t_2]$
Join: $\bowtie$, with a join condition c • $t \in R_{\mathcal{P}_1} \bowtie_c S_{\mathcal{P}_2}$ iff $t \in R_{\mathcal{P}_1} \times S_{\mathcal{P}_2}$ and satisfies c • $t_1 <_{R_{\mathcal{P}_1}\bowtie_c S_{\mathcal{P}_2}} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_2]$

Fig. 13. Operators defined in RankSQL algebra [Li et al. 2005].

4.2.2. Query Algebra. Formalizing the interaction between ranking and other relational operations, for example, selections and joins, through an algebraic framework, gives more potential to optimize top-k queries. Taking the ranking requirements into account, while building a top-k query plan, has been shown to yield significant performance gains compared to the conventional materialize-then-sort techniques [Ilyas et al. 2004b]. These ideas are the foundations of the RankSQL system [Li et al. 2005], which introduces the first algebraic framework to support efficient evaluations of top-k queries in relational database systems.

RankSQL views query ranking requirements as a *logical property*, similar to the conventional membership property. That is, each base or intermediate relation (the relation generated by a query operator during query execution) is attributed with the base relations it covers (the membership property), as well as the tuple orders it provides (the order property). RankSQL extends traditional relational algebra by introducing a new algebra that embeds rank-awareness into different query operators. The extended rank-relational algebra operates on *rank-relations*, which are conventional relations augmented with an *order property*.

Figure 13 summarizes the definitions of the rank-aware operators in RankSQL. The symbol $R_{\mathcal{P}}$ denotes a rank-relation R whose order property is the set of ranking predicates $\mathcal{P}$. The notation $\overline{\mathcal{F}}_{\mathcal{P}}[t]$ denotes the upper bound of the scoring function $\mathcal{F}$ for a tuple t, based on a set of ranking predicates $\mathcal{P}$. This upper bound is obtained by applying $\mathcal{F}$ to $\mathcal{P}$ and the maximum values of all other scoring predicates not included in $\mathcal{P}$.

A new rank-augment operator μ_p is defined in RankSQL to allow for *augmenting* the order property of some rank-relation with a new scoring predicate p. That is, $\mu_p(R_{\mathcal{P}}) = R_{\mathcal{P}\cup\{p\}}$. The semantics of the relational operators π, σ, $\cup$, $\cap$, $-$, and $\bowtie$ are extended to add awareness of the orders they support. Figure 13 shows how the membership and order properties are computed for different operators. For unary

operators, such as π and σ, the same tuple order in their inputs is maintained. That is, only the membership property can change, for example, based on a Boolean predicate, while the order property remains the same. On the other hand, binary operators, such as $\cap$ and $\bowtie$, involve both Boolean predicates and order *aggregation* over their inputs, which can change both the membership and order properties. For example, $R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2} \equiv (R \cap S)_{\mathcal{P}_1 \cup \mathcal{P}_2}$. We discuss the optimization issues of such rank-aware operators in Section 4.2.3.

4.2.3. Query Optimization. Rank-aware operators need to be integrated with query optimizers to be practically useful. Top-k queries often involve different relational operations such as joins, selections, and aggregations. Building a query optimizer that generates efficient query plans satisfying the requirements of such operations, as well as the query ranking requirements, is crucial for efficient processing. An observation that motivates the need for integrating rank-aware operators within query optimizers is that using a rank-aware operator may not be always the best way to produce the required ranked results [Ilyas et al. 2004b, 2006]. The reason is that there are many parameters (e.g., join selectivity, available access paths, and memory size) that need to be taken into account while comparing rank-aware and conventional query operators.

Integrating rank-aware and conventional operators in query optimization has been mainly addressed form two perspectives. In the first perspective, the plan enumeration phase of the query optimizer is extended to allow for mixing and interleaving rank-aware operators with convectional operators, creating a rich space of different top-k query plans. In the second perspective, cost models for rank-aware operators are used by the query optimizer to compare the expected cost of rank-aware operators against conventional operators, and hence construct efficient query plans.

4.2.3.1. Plan Enumeration. The RankSQL system [Li et al. 2005] extends the dynamic programming plan enumeration algorithm, adopted by most RDBMSs, to treat ranking as a *logical property*. This extension adds an extra enumeration dimension to the conventional membership dimension. In a ranking query plan, the set of scoring predicates in a query subplan determines the order property, just like how the join conditions (together with other operations) determine the membership property. This allows the enumerator to produce different equivalent plans, for the same logical algebra expression, where each plan interleaves ranking operators with other operators in a different way. The two-dimensional (order + membership) enumeration algorithm of RankSQL maintains membership and order properties for each query subplan. Subplans with the same membership and order properties are compared against each other, and the inferior ones, based on cost estimation, are pruned.

Figure 14 illustrates the transformation laws of different query operators in RankSQL. The transformations encapsulate two basic principles: (1) splitting (Proposition 1), where ranking is evaluated in stages, predicate by predicate, and (2) interleaving (Propositions 4 and 5), where Ranking is interleaved with other operators. Splitting rank expression allows embedding rank-aware operators in the appropriate places within the query plan, while interleaving guarantees that tuples are pipelined in the correct order such that tuple flow can be stopped as soon as the top-k results are generated. The next example illustrates the above principles.

Example 4.1 (*RankSQL Example*). Consider the following top-k query:

SELECT * FROM R, S
WHERE $R.j = S.j$
ORDER BY $R.p_1 + S.p_1 + R.p_2$
LIMIT 10

Proposition 1: Splitting law for μ • $R_{\{p_1,p_2,\ldots,p_n\}} \equiv \mu_{p_1}(\mu_{p_2}(\ldots(\mu_{p_n}(R))\ldots))$
Proposition 2: Commutative law for binary operator • $R_{\mathcal{P}_1} \Theta S_{\mathcal{P}_2} \equiv S_{\mathcal{P}_2} \Theta R_{\mathcal{P}_1}, \forall \Theta \in \{\cap, \cup, \bowtie_c\}$
Proposition 3: Associative law • $(R_{\mathcal{P}_1} \Theta S_{\mathcal{P}_2}) \Theta T_{\mathcal{P}_3} \equiv R_{\mathcal{P}_1} \Theta (S_{\mathcal{P}_2} \Theta T_{\mathcal{P}_3}), \forall \Theta \in \{\cap, \cup, \bowtie_c{}^a\}$
Proposition 4: Commutative laws for μ • $\mu_{p_1}(\mu_{p_2}(R_{\mathcal{P}})) \equiv \mu_{p_2}(\mu_{p_1}(R_{\mathcal{P}}))$ • $\sigma_c(\mu_p(R_{\mathcal{P}})) \equiv \mu_p(\sigma_c(R_{\mathcal{P}}))$
Proposition 5: Pushing μ over binary operators • $\mu_p(R_{\mathcal{P}_1} \bowtie_c S_{\mathcal{P}_2})$ $\equiv \mu_p(R_{\mathcal{P}_1}) \bowtie_c S_{\mathcal{P}_2}$, *if only* R *has attributes in* p $\equiv \mu_p(R_{\mathcal{P}_1}) \bowtie_c \mu_p(S_{\mathcal{P}_2})$, *if both* R *and* S *have* • $\mu_p(R_{\mathcal{P}_1} \cup S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cup \mu_p(S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cup S_{\mathcal{P}_2}$ • $\mu_p(R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cap \mu_p(S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cap S_{\mathcal{P}_2}$ • $\mu_p(R_{\mathcal{P}_1} - S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) - S_{\mathcal{P}_2} \equiv \mu_p(R_{\mathcal{P}_1}) - \mu_p(S_{\mathcal{P}_2})$
Proposition 6: Multiple-scan of μ • $\mu_{p_1}(\mu_{p_2}(R_\phi)) \equiv \mu_{p_1}(R_\phi) \cap_r \mu_{p_2}(R_\phi)$

[a]When join columns are available.

Fig. 14. RankSQL algebraic laws [Li et al. 2005].

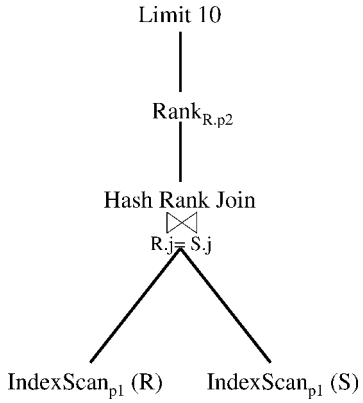

Fig. 15. A query plan generated by RankSQL.

Figure 15 depicts a query plan generated by RankSQL, where ranking expression is split and rank-aware operators are interleaved with conventional query operators. The plan shows that the query optimizer considered using existing indexes for $R.p_1$ and $S.p_1$ to access base tables, which might be cheaper for the given query, since they generate tuples in the orders of scoring predicates $R.p_1$ and $S.p_1$. The *Hash Rank Join* operator is a rank-aware join operator that aggregates the orders of the joined relations, while the *Rank* operator augments the remaining scoring predicate $R.p2$ to produce tuples in the required order. The *Hash Rank Join* and *Rank* operators pipeline their outputs by upper bounding the scores of their unseen inputs, allowing for consuming a small number of tuples in order to find the top-10 query results. Figure 16 depicts an

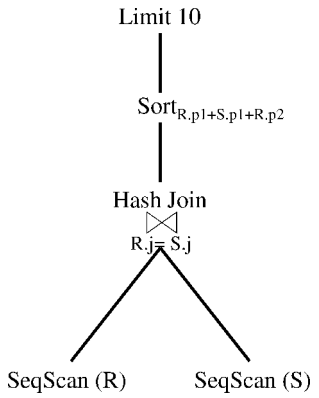

Fig. 16. Conventional query plan.

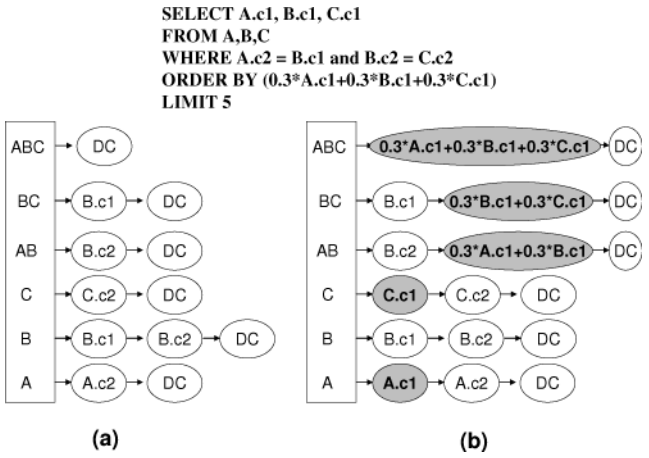

Fig. 17. Plan enumeration (a) in conventional optimizer, and (b) with ordering as an *interesting property* [Ilyas et al. 2004b].

equivalent conventional plan, where the entire query result is materialized and sorted on the given ranking expression, then the top-10 results are reported.

Treating ranking requirements as a *physical property* is another approach to extend the plan enumerator. A physical property is a plan characteristic that can be different in different plans with the same membership property, but impacts the cost of subsequent operations. Ilyas et al. [2004b, 2006] defined ranking requirements as an *interesting physical order*, which triggers the generation of new query plans to optimize the use of Rank-Join operators. An interesting order is an order of query intermediate results, based on an *expression* of database columns, that can be beneficial to subsequent query operations. An interesting order can thus be the order of a join column or the order of a scoring predicate. To illustrate, Figure 17(a) shows the plan space maintained by a *conventional optimizer* for the shown three-way join query. In the shown MEMO structure, each row represents a set of plans having the same membership property, while each plan maintains a different interesting order. The plan space is generated by comparing plans with the same interesting order at

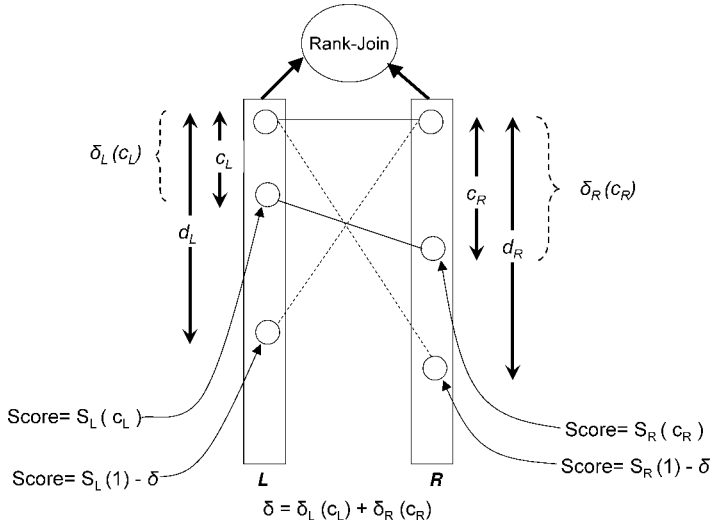

Fig. 18. Estimating depth of Rank-Join inputs.

each row, and keeping the plan with the lowest cost. The node DC refers to a *don't care* property value, which corresponds to no order. For example, plans in the B row provide tuples of the B relation (the membership property) ordered by $B.c1$, $B.c2$, and no order. These orders are interesting since they are included in the query join conditions.

Consider alternatively Figure 17 (b), which shows the plan space generated by treating the orders of scoring predicates as additional interesting orders. The shaded nodes indicate the interesting orders defined by scoring predicates. Enumerating plans satisfying the orders of scoring predicates allows generating rank-aware join choices at each step of the plan enumeration phase. For example, on the level of base tables, the optimizer enforces the generation of table scans and index scans that satisfy the orders of the scoring predicates. For higher join levels, the enumerated access paths for base tables make it feasible to use Rank-Join operators as join choices.

4.2.3.2. Cost Estimation. The performance gain obtained by using rank-aware query operators triggers thinking of more principled approaches to integrate ranking with conventional relational operators. This raises the issue of cost estimation. Query optimizers need to be able to enumerate and cost plans with rank-aware operators as well as conventional query operators. The cost model of query operators depends on many parameters including input size, memory buffers and access paths. The optimization study of the Rank-Join operator [Ilyas et al. 2004b, 2006] shows that costing a rank-aware operator is quite different from costing other traditional query operators. A fundamental difference stems from the fact that a rank-aware operator is expected to consume only part of its input, while a traditional operator consumes its input completely. The size of the consumed input depends on the operator implementation rather than the input itself.

A probabilistic model has been proposed in Ilyas et al. [2004b] to estimate the rank-join inputs' depths, that is, how many tuples are consumed from each input to produce the top-k join results. Figure 18 depicts the depth estimation procedure. For inputs L and R, the objective is to get the estimates d_L and d_R such that it is sufficient to retrieve only up to d_L and d_R tuples from L and R, respectively, to produce the top-k

join results. The definitions of other used notations are listed in the following:

—c_L and c_R are depths in L and R, respectively, that are sufficient to find any k valid join results. c_L and c_R can be selected arbitrarily such that $s.c_L.c_R \geq k$, where s is the join selectivity of L joined with R.

—$S_L(i)$ and $S_R(i)$ are the scores at depth i in L and R, respectively.

—$\sigma_L(i)$ and $\sigma_R(i)$ are the score differences between the top ranked tuple and the the tuple at depth i in L and R, respectively.

—$\sigma = \sigma_L(c_L) + \sigma_R(c_R)$.

The main result of Ilyas et al. [2004b] is the following: if d_L and d_R are selected such that $\sigma_L(d_L) \geq \sigma$ and $\sigma_R(d_R) \geq \sigma$, then the top-$k$ join results can be obtained by joining $L(d_L)$ and $R(d_R)$. Further analysis based on the score distribution in L and R has been conducted to reach the minimum possible values for d_L and d_R. For uniformly distributed scores of L and R with average score slabs (average distance between two consecutive scores) of x and y, respectively, the expected value of $\sigma_L(c_L) = x.c_L$ and the expected value of $\sigma_R(c_R) = y.c_R$. Hence, to minimize d_L and d_R, we need to minimize $\sigma = \sigma_L(c_L) + \sigma_R(c_R) = x.c_L + y.c_R$ subject to $s.c_L.c_R \geq k$. A direct way to minimize this expression is to select $c_L = \sqrt{(yk)/(xs)}$ and $c_R = \sqrt{(xk)/(ys)}$.

5. QUERY AND DATA UNCERTAINTY

In this section, we discuss top-k processing techniques that report approximate answers while operating on deterministic data (Section 5.1), followed by techniques that operate on probabilistic data (Section 5.2).

5.1. Deterministic Data, Approximate Methods

Reporting the exact top-k query answers could be neither cheap, nor necessary for some applications. For example, decision support and data analysis applications usually process huge volumes of data, which may cause significant delays if the application requires exact query answers. Users in such environments may thus sacrifice the accuracy of query answers in return of savings in time and resources. In these settings, reporting approximate query answers could be sufficient.

We start by describing some of the general approximate query processing techniques, followed by approximate top-k processing techniques.

5.1.1. Approximate Query Processing. The work of Vrbsky and Liu [1993] is one of the earliest attempts made to approximate query answers. The proposed system, called *APPROXIMATE*, is a query processor that produces approximate answers that enhance monotonically. In this context, an operation is monotone if its results are more accurate when its operands are more accurate. APPROXIMATE works on *approximate relations* that are defined as subsets of the cartesian product of attribute domains. The tuples in these subsets are partitioned into *certain* and *possible* tuples. Approximate relations are used during query processing instead of the standard relations generated by different query tree nodes.

Based on the semantics of the data stored in the base tables, an initial approximation to query answers is driven. This initial approximation is assigned to each node in the query tree. As more certain data is read from base tables, more certain tuples are inserted into approximate relations while more possible tuples are deleted. If a query is stopped before completion, a superset of the exact answer is returned. One problem with the proposed method is that it mainly depends on detailed metadata

describing the semantics of database relations, which might not be easy to construct in practice.

Multiple approximate query processing methods have addressed aggregate queries. Aggregate queries are widely used in OLAP environments for data exploration and analysis purposes. In this kind of queries, the precision of the answer to the last decimal digit is not usually needed. The framework of *Online Aggregation* [Hellerstein et al. 1997] addresses generating approximate aggregate values whose accuracy enhances progressively. The work of Hellerstein et al. [1997] provides important insights on using statistical methods in aggregate computation. The adopted statistical methods allow computing aggregate values along with a correctness probability and confidence interval derived based on the expected aggregate value.

Other approximate query processing methods involve building a database summary or synopsis, and using it to answer queries approximately. Sampling is probably the most studied methodology for building such summaries. The basic settings involve drawing a sample from the underlying data, answering the incoming queries based on the sample, and scaling the results to approximate the exact answers. Sampling can be done either uniformly at random or in a biased manner to select the best samples that minimize the approximation errors [Chaudhuri et al. 2001b]. Other possible summaries include histograms [Babcock et al. 2003; Chaudhuri et al. 1998] and wavelets Chakrabarti et al. [2001].

Approximation based on sampling has inherent problems that negatively effect the accuracy in many cases. One problem is the presence of skewness in data distributions. Skewed data items, usually called *outliers*, are those items which deviate significantly from other data items in their values. Outliers cause a large variance in the distribution of aggregate values, which leads to large approximation errors. Separating outliers using a special index is one approach to deal with this problem [Chaudhuri et al. 2001]. In this setting, queries are considered as the union of two subqueries, one of which is answered exactly using an outlier index, while the other is answered approximately by sampling the nonoutliers. The two results are then combined to give the full approximate query answer.

Another problem is the potentially low selectivity of selection queries. Uniform samples contain only a small fraction of the answers of highly selective queries. Nonuniform sampling is proposed to work around this problem. The idea is to use *weighted sampling*, where tuples are tagged by their frequency—the number of workload queries whose answers contain the tuple. Tuples that are accessed frequently by previous queries would therefore have higher probability to be included in the sample. The underlying assumption is that tuples that are part of the answers to previous queries are likely to be part of the answers to similar incoming queries. Collecting samples offline based on previous queries, and rewriting incoming queries to use these samples, were proposed in Chaudhuri et al. [2001a]. Self-tuning the samples by refreshing them after queries are processed was further studied by Ganti et al. [2000].

The presence of small groups in *group-by* queries could also lead to large approximation errors when using sampling techniques. *Congressional samples* [Acharya et al. 2000] addressed this problem by introducing a hybrid of uniform and nonuniform samples. The proposed strategy is to divide the available sample space equally among the groups, and take a uniform random sample within each group. This guarantees that both large and small groups will have a reasonable number of samples. The problem with this approach is that it does not deal with the data variance caused by outliers.

5.1.2. Approximate Top-k Query Processing. Adopting the concepts and techniques of approximate query processing in the context of top-k queries is a natural extension of

the previously described works. Some general top-k algorithms, for example, TA [Fagin et al. 2001], have approximate variants. For example, an approximate variant of TA defines a parameter $\theta > 1$ denoting the required level of approximation, such that an item z not in the top-k set satisfies the condition $score(z) \leq \theta score(y)$ for every other item y inside the top-k set. This approximation relaxes the threshold test of TA, making it possible to return approximate answers. However, the problem with this approach is that the selection of the approximation parameter θ is mostly application-oriented and that no general scheme is devised to decide its value. Another example is the J^* algorithm [Natsev et al. 2001] that allows approximating tuple ranks to a factor $0 < \epsilon < 1$. The ϵ-approximate top-k answer, X, is defined as $\forall x \in X, y \notin X : (1+\epsilon)x.score \geq y.score$. When $\epsilon = 0$, the approximation algorithm reduces to the exact one.

Approximate answers are more useful when they are associated with some accuracy guarantees. This issue has been addressed by another approximate adaptation of TA [Theobald et al. 2004], where a scheme was introduced to associate probabilistic guarantees with approximate top-k answers. The proposed scheme works in the context of information retrieval systems, assuming multiple lists each holding a different ranking of an underlying document set based on different criteria, such as query keywords. The conservative TA threshold test is replaced by a probabilistic test to estimate the probability that a candidate document would eventually be in the top-k set. Specifically, the probability of document d to have a score above M_k, the minimum score of the current top-k set, is computed as follows:

$$Pr\left(\sum_{i \in E(d)} p_i(d) + \sum_{j \notin E(d)} \hat{p}_j(d) > M_k\right), \tag{3}$$

where $E(d)$ is the set of lists in which d has been encountered, $p_i(d)$ is the score of document d in list i, and $\hat{p}_j(d)$ is an estimator for the score of d in list j, where it has not been encountered yet. If this probability is below a threshold ϵ, the document d is discarded from the candidate set. Setting $\epsilon = 0$ corresponds to the conservative TA test. The main departure from standard TA is that unseen partial scores are estimated probabilistically instead of setting them, loosely, to the best possible unseen scores. Computing the estimators of unseen partial scores is based on the score distributions of the underlying ranked lists. Three different distributions are considered: uniform, Poisson, and a generic distribution derived from existing histograms.

The above approximation technique is illustrated by Algorithm 9. For each index list, the current position and last seen score are maintained. For each item d that is encountered in at least one of the lists, a worst score $\underline{F}(d)$ is computed by assuming zero scores for d in lists where it has not been encountered yet, and a best score $\overline{F}(d)$ is computed by assuming the highest possible scores of unseen items in the lists where d has not been encountered yet. The minimum score among the current top-k items, based on items' worst scores, is stored in M_k. An item is considered a *candidate* to join the top-k set if its best score is above M_k. Periodically, the candidate set is filtered from items whose best scores cannot exceed M_k anymore, or items that fail the probabilistic test of being in the top-k . This test is performed by computing the probability that item's total score is above M_k. If this probability is below the approximation threshold ϵ, then the item is discarded from the candidates without computing its score.

Algorithm 9. Top-k with Probabilistic Guarantees [Theobold et al. 2004]

```
1: topk = {dummy_1, ... dummy_k} with F(dummy_i) = 0
2: M_k = 0          {the minimum score in the current topk set}
3: candidates = φ
4: while index lists L_i (i = 1 to m) are not exhausted do
5:       d = next item from L_i
6:       p̄_i = p_i(d)
7:       add i to E(d)
8:       set F̲(d) = Σ_{r∈E(d)} p_r(d), and F̄(d) = F̲(d) + Σ_{r∉E(d)} p̄_r
9:       if F̄(d) > M_k then
10:            add d to candidates
11:      else
12:            drop d from candidates if present
13:   end if
14:      if F̲(d) > M_k then
15:            if d ∉ topk then
16:                  remove the item d' with the min F̲(.) in topk
17:                  add d to topk
18:                  add d' to candidates
19:         end if
20:         M_k = min{F̲(d')|d' ∈ topk}
21:   end if
22:      periodically do the following loop
23:      for all d' ∈ candidates do
24:            update F̄(d')
25:            P = Pr(Σ_{i∈E(d')} p_i(d') + Σ_{j∉E(d')} p̂_j(d') > M_k)
26:            if F̄(d') < M_k or P < ε then
27:                  drop d' from candidates
28:         end if
29:   end for
30:      T = max{F̄(d')|d' ∈ candidates}
31:      if candidates = φ or T ≤ M_k then
32:            return topk
33:   end if
34: end while
```

Reporting approximate top-k answers is also considered in similarity search in multimedia databases. A similarity search algorithm usually uses a distance metric to rank objects according to their distance from a target query object. In higher dimensions, a query can be seen as a hypersphere, centered at the target object. The number of data objects that intersect the query sphere depends on the data distribution, which makes it possible to apply probabilistic methods to provide approximate answers [Amato et al. 2003]. In many cases, even if the query region overlaps a data region, no actual data points (or a small number of data points) appear in the intersection, depending on data distribution. The problem is that there is no way to precisely determine the useless regions, whose intersection with the query region is empty, without accessing such regions. The basic idea in Amato et al. [2003] is to use a *proximity* measure to decide if a data region should be inspected or not. Only data regions whose proximity to the query region is greater than a specified threshold are accessed. This method is used to rank the nearest neighbors to some target data object in an approximate manner.

Approximate top-k query processing has been also studied in peer-to-peer environments. The basic settings involve a query initiator submitting a query (mostly in the form of keywords) to a number of sites that respond back with the top-k answers, based on their local scores. The major problem is how to efficiently coordinate the communication among the respondents in order to aggregate their rankings. The KLEE system [Michel et al. 2005] addresses this problem, where distributed aggregation queries are processed based on index lists located at isolated sites. KLEE assumes no random accesses are made to index lists located at each peer. Message transfers among peers are reduced by encoding messages into lightweight Bloom filters representing data summaries.

The query processing in KLEE starts by exploring the network to find an initial approximation for the minimum score of the top-k set (M_k). The query initiator sends to each peer M_k/k as an estimate for the minimum score of the top-k results expected from that peer. When a peer receives a query, it finds the top-k items locally, builds a histogram on their scores, and hashes the items' IDs in each histogram cell into a Bloom filter. Each peer returns back to the initiator a Bloom filter containing the IDs of all items with scores above M_k/k. The query initiator combines the received answers, extracts the top-k set, and inserts the remaining items into a candidate set. Candidate items are filtered by identifying the IDs of items with high scores. The structure of Bloom filters facilitates spotting the items that are included in the answers of a large number of peers. The good candidates are finally identified and requested from the peers.

5.2. Uncertain Data

Uncertain data management has gained more visibility with the emergence of many practical applications in domains like sensor networks, data cleaning, and location tracking, where data is intrinsically uncertain. Many uncertain (probabilistic) data models, for example, Fuhr [1990], Barbará et al. [1992], and Imielinski and Lipski Jr. [1984], have been proposed to capture data uncertainty on different levels. According to many of these models, tuples have membership probability, for example, based on data source reliability, expressing the belief that they should belong to the database. A tuple attribute could also be defined probabilistically as multiple possible values drawn from discrete or continuous domains, for example, a set of possible customer names in a dirty database, or an interval of possible sensor readings.

Many uncertain data models adopt *possible worlds* semantics, where an uncertain database is viewed as a set of possible instances (worlds) associated with probabilities. Each possible world represents a valid combination of database tuples. The validity of some tuple combination is determined based on the underlying tuple dependencies. For example, two tuples might never appear together in the same world if they represent the same real-world entity. Alternatively, the existence of one tuple in some world might imply the existence of another tuple.

Top-k queries in deterministic databases assume a single ranking dimension, namely, tuples' scores. In probabilistic databases, tuples' probabilities arise as an additional ranking dimension that interacts with tuples' scores. Both tuple *probabilities* and *scores* need to be factored in the interpretation of top-k queries in probabilistic databases. For example, it is not meaningful to report a top-scored tuple with insignificant probability. Alternatively, it is not accepted to order tuples by probability, while ignoring their scores. Moreover, combining scores and probabilities using some score aggregation function eliminates uncertainty completely, which may not be meaningful in some cases, and does not conform with the currently adopted probabilistic query models.

The scores-uncertainty interaction of top-k queries in probabilistic databases was addressed in Soliman et al. [2007], where a processing framework was introduced to

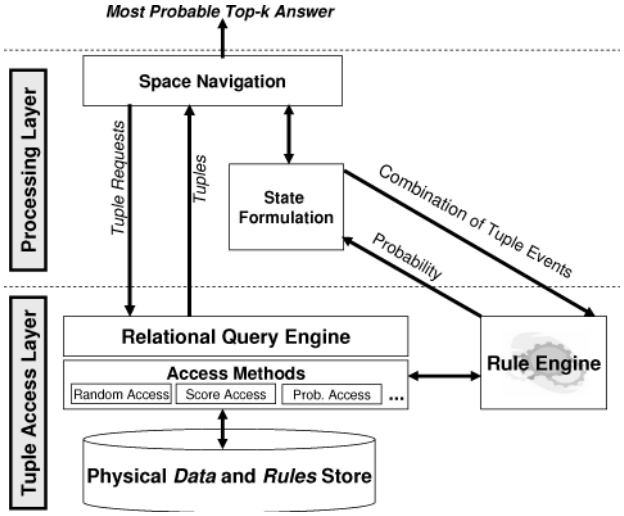

Fig. 19. Processing framework for uncertain top-k processing [Soliman et al. 2007].

find the most probable top-k answers in uncertain databases. The interaction between the concepts of "most probable" and "top-k" is materialized using two new top-k query semantics: (1) U-Top$k$ query: a top-k query that reports a k-length tuple vector with the maximum probability of being top-k across all database possible worlds; and (2) U-kRanks query: a top-k query that reports a set of k tuples, where each tuple is the most probable tuple to appear at some rank $1 \cdots k$ across all database possible worlds. These two interpretations involve both ranking and aggregation of possible worlds.

Figure 19 depicts the processing framework introduced in Soliman et al. [2007] to answer top-k queries in uncertain databases. The framework leverages RDBMS storage, indexing, and query processing techniques, in addition to probabilistic inference tools, to compute the most probable top-k answers. The framework contains two main layers, described in the following:

—*Tuple access layer*. Tuple retrieval, indexing, and traditional query processing (including score-based ranking) are the main functionalities provided by this layer. Uncertain data and probabilistic dependencies are stored in a relational database with different access methods provided to allow the processing layer to retrieve the uncertain tuples.

—*Processing layer*. The processing layer retrieves uncertain tuples from the underlying storage layer, and efficiently navigates the space of possible worlds to compute the most probable top-k answers.

The problem of finding top-k query answers in uncertain databases is formulated as searching the space of *states* that represent possible top-k answers, where a state is a possible *prefix* of one or more worlds ordered on score. Each state has a probability equal to the aggregate probability of the possible worlds prefixed by this state. The *Rule Engine* component in Figure 19 is responsible for computing such probabilities using probabilistic inference tools, for example, Bayesian networks. The search for uncertain top-k answers starts from an empty state with length 0 and ends at a goal state with length k, having the maximum probability. The proposed search algorithms minimize the number of accessed tuples, and the number of visited space states.

The problem of finding the k most probable query answers in probabilistic databases was addressed by Ré et al. [2007]. In this setting, *probability* is the only ranking dimension, since tuples are not scored by a scoring function. Tuples are treated as probabilistic events with the assumption that base tuples correspond to independent events. However, when relational operations, for example, joins and projections, are conducted on base tuples, the independence assumption does not hold any more on the output tuples. Computing the exact probabilities of the answers in this case is generally in #P-complete, which is the class of counting algorithms corresponding to the NP-complete complexity class.

To address the above challenges, Ré et al. [2007] proposed a multisimulation algorithm (MS_Topk) based on Monte-Carlo simulation. In MS_Topk, computing the exact probability of an answer is relaxed in favor of computing the correct ranking efficiently. MS_Topk maintains a probability interval for each candidate top-k answer enclosing its exact probability. In each step, the probability intervals of some candidates are tightened by generating random possible worlds, and testing whether these candidates belong to such worlds or not. The probability intervals of candidate answers are progressively tightened until there are k intervals with no other intervals overlapping with their minimum bounds, and hence the top-k answers are obtained. Ré et al. [2007] also proposed other variants of the MS_Topk algorithm to sort the top-k answers and incrementally report the top-k answers one by one.

6. RANKING FUNCTION

In this section, we discuss the properties of different ranking functions adopted by top-k processing techniques. Top-k processing techniques are classified into three categories based on the type of ranking functions they assume. The first category, which includes the majority of current techniques, assumes *monotone* ranking functions. The second category allows for *generic* ranking functions. The third category leaves the ranking function unspecified.

6.1. Monotone Ranking Functions

The majority of top-k techniques assumes monotone scoring functions, for example, TA [Fagin et al. 2001], and UPPER [Bruno et al. 2002b]. Using monotone ranking functions is common in many practical applications, especially in Web settings [Marian et al. 2004]. For example, many top-k processing scenarios involve linear combinations of multiple scoring predicates, or maximum/minimum functions, which are all monotone.

Monotone ranking functions have special properties that can be exploited for efficient processing of top-k queries. As demonstrated by our previous discussion of many top-k processing techniques, when aggregating objects' scores from multiple ranked lists using a monotone score aggregation function, an upper bound of the scores of unseen objects is easily derived. To illustrate, assume that we sequentially scan m sorted lists, $L_1, \ldots, L_m$, for the same set of objects based on different ranking predicates. If the scores of the last retrieved objects from these lists are $\overline{p}_1, \overline{p}_2, \ldots, \overline{p}_m$, then an upper bound, $\overline{F}$, over the scores of all unseen objects is computed as $\overline{F} = F(\overline{p}_1, \overline{p}_2, \ldots, \overline{p}_m)$. It is easy to verify that no unseen object can possibly have a score greater than the above upper bound $\overline{F}$; by contradiction, assume that an unseen object o_u has a score greater than $\overline{F}$. Then, based on F's monotonicity, o_u must have at least one ranking predicate p_i with score greater than the last seen score $\overline{p}_i$. This implies that the retrieval from list L_i is not score-ordered, which contradicts with the original assumptions. Additionally, monotone scoring functions allow computing a score upper bound for each seen object o

by substituting the values of unknown ranking predicates of o with the last seen scores in the corresponding lists. A top-k processing algorithm halts when there are k objects whose score lower bounds are not below the score upper bounds of all other objects, including the unseen objects.

The above properties are exploited in various algorithms, for example, TA [Fagin et al. 2001] and Quick-Combine [Güntzer et al. 2001] (discussed in Section 3.1), to guarantee early termination of top-k processing. An important property that follows from monotonicity is that these algorithms are *instance-optimal*, within some bounds, in the sense that, for any database instance, there is no other algorithm that can retrieve less number of objects and return a correct answer. For proof of instance optimality, we refer the reader to Fagin et al. [2001].

Linear ranking functions constitute a subset of monotone functions. Linear ranking functions define the aggregated score of ranking predicates as a weighted sum. Several top-k techniques exploit the geometrical properties of linear functions to efficiently retrieve the top-k answers, such as the *Onion Indices* [Chang et al. 2000] and *Ranked Join Indices* [Tsaparas et al. 2003], discussed in Section 4.1.2. Linear functions can be represented as vectors based on the weights associated with the ranking predicates. Such representation can be used to geometrically compute objects' scores as projections on the ranking function vector. Other properties such as the relation between linear functions and convex hulls of the data points are also exploited as in Chang et al. [2000].

6.2. Generic Ranking Functions

Using nonmonotone ranking functions in top-k queries is challenging. One reason is that it is not straightforward to prune objects that do not qualify to query answer at an early stage. The methods used to upper-bound the scores of unseen (or partially seen) objects are not applicable to nonmonotone functions. For example, assume two sorted lists based on ranking predicates p_1 and p_2, and a scoring function $F(t) = p_1(t)/p_2(t)$. The function F in this case is obviously nonmonotone. The last seen scores $\overline{p}_1$ and $\overline{p}_2$ cannot be used to derive a score upper bound for the unseen objects. The reason is that it is possible to find objects with scores above $\overline{p}_1/\overline{p}_2$ later on if the unseen scores in the first list are all equal to $\overline{p}_1$, while the unseen scores in the second list, p_2, decrease.

Some recent proposals have addressed the challenges imposed by generic (not necessarily monotone) ranking functions. The technique proposed in Zhang et al. [2006] supports arbitrary ranking functions by modeling top-k query as an optimization problem. The optimization goal function consists of a Boolean expression that filters tuples based on query predicates, and a ranking function that determines the score of each tuple. The goal function is equal to zero whenever a tuple does not satisfy the Boolean expression, and it is equal to the tuple's score otherwise. The answer to the top-k query is the set of k tuples with the highest values of the goal function.

In order to efficiently search for the top-k answers, existing indexes of scoring predicates are used. The optimization problem is solved using an $\mathcal{A}^*$ search algorithm, named *OPT**, by transforming the problem into a shortest path problem as follows. Each state in the search space is created by joining index nodes. A state thus covers subsets of predicate domains. Two types of states are defined at different index levels. The first type is *region states*, which represents internal index nodes, while the second type is *tuple states*, which represents index leaf nodes (i.e., the tuple level of the index). A transition between two states is achieved by traversing the indexes. A dummy *goal* state is created so that it can be reached from any tuple state, where the distance from a tuple state to the goal state is equal to the inverse of tuple's score. Distances between other state pairs are set to zero. Based on this formulation, the shortest path to the goal state involves the tuples with the maximum scores. To reach the goal state in a small number

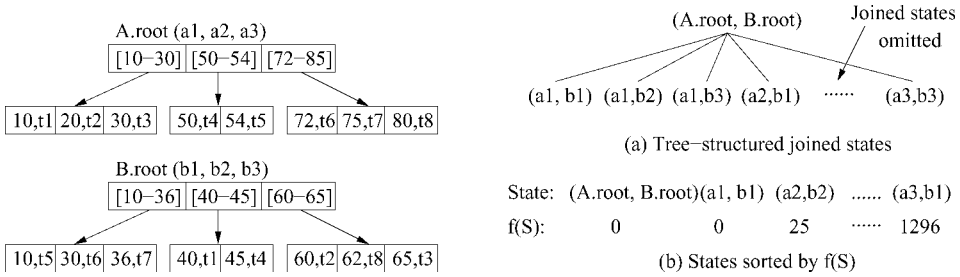

Fig. 20. Index Merge [Xin et al. 2007]. (a) Indices on predicates A and B; (b) space of joint states.

of state transitions, a heuristic function is used to guide transitions using the maximum possible score of the current state. The authors proved that the OPT* algorithm is optimal in the number of visited leaves, tuples, and internal states under certain constraints over the weights associated with each term in the optimization function.

Ad hoc ranking functions were addressed by Xin et al. [2007], with the restriction that the function is lower-bounded. A ranking function F is lower-bounded in a region Ω of its variables domains, if the lower bound of F in Ω can be derived. Examples include $F = (x - y)^2$. The authors presented an index-merge framework that performs progressive search over a space of states composed by joining index nodes. The main idea is to exploit existing B-Tree and R-Tree indexes of ranking predicates to create a search space of possible query answers. A state in this space is composed by joining multiple index nodes. Promising search states, in terms of their involved scores, are progressively materialized, while the states with no chances of yielding the top-k answers are early-pruned. The search algorithm prioritizes visiting space states based on their score lower bounds. For each state, an internal heap is maintained to prioritize traversing the state's children (subspaces).

To understand the search space of the above technique, consider Figure 20, which shows the search space for a ranking function $f = (A - B)^2$, where A and B are two ranking predicates. Figure 20(a) shows two existing indices on predicates A and B, while Figure 20(b) shows the generated search space, and the computed score lower bounds of different states. Let $I1$ and $I2$ be the two indices defined on predicates A and B, respectively. The search space formed by joining $I1$ and $I2$ is constructed based on the hierarchical index structure. That is, for any joint state $(I1.n1, I2.n2)$, its child states are created as the Cartesian products of child nodes of $I1.n1$ and $I1.n2$. For example, in Figure 20(a), the joint state $(A.root, B.root)$ has the following children: $\{(a1, b1), (a1, b2), (a1, b3), (a2, b1), (a2, b2), (a2, b3), (a3, b1), (a3, b2), (a3, b3)\}$. Score lower bounds of different states are used to order state traversal. For example, the state $(a2, b2)$ has a score lower bound of 25, which is computed based on the square of the difference between 50 and 45, which are possible tuple scores in index nodes $a2$ and $b2$, respectively.

6.3. No Ranking Function (Skyline Queries)

In some applications, it might not be straightforward to define a ranking function. For example, when query answers are to be returned to more than one user, it might be more meaningful to report all "interesting" answers, rather than a strict ranking according to some specified ranking function.

A *skyline* query returns the objects that are not dominated by any other objects restricted to a set of dimensions (predicates). Figure 21 shows the skyline of a number of objects based on two predicates p_1 and p_2, where larger values of both predicates

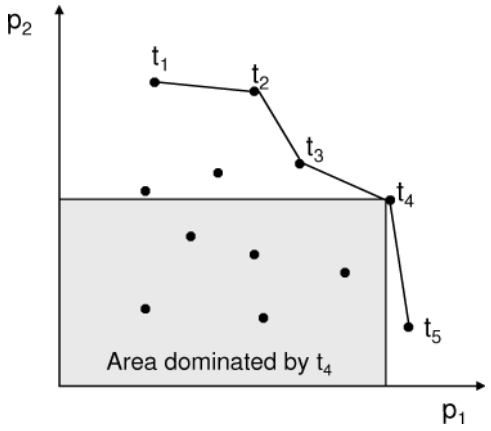

Fig. 21. Skyline query.

are more favored. An object X dominates object Y if ($X.p_1 > Y.p_1$ and $X.p_2 \geq Y.p_2$) or ($X.p_1 \geq Y.p_1$ and $X.p_2 > Y.p_2$). For example t_4 dominates all objects in the shaded rectangle in Figure 21. All the objects that lie on the skyline are not dominated by any other object. An interesting property of the skyline is that the top-1 object, based on any monotone ranking function, must be one of the skyline objects. The *Onion Indices* [Chang et al. 2000] make use of this property by materializing and indexing the k first skylines and then answering the query for some ranking function by only searching the skyline objects.

There is a large body of research that addresses skyline related queries. Details of skyline query processing techniques are out of the scope of this survey.

7. TOP-*K* QUERY PROCESSING IN XML DATABASES

Top-k processing in XML databases has recently gained more attention since XML has become the preferred medium for formatting and exchanging data in many domains such as the Web and e-commerce. Top-k queries are dominant type of queries in such domains.

In XML queries, *structure* and *content* constraints are usually specified to express the characteristics of the required query answers. XML elements that (partially) match query conditions are usually associated with relevance scores capturing their similarity to the query conditions. It is often required to rank query matches based on their relevance scores, or report only the top-k matches. There are two main issues that need to be addressed in this type of queries. The first issue is what kind of scoring functions can be used to measure the relevance of answers with respect to query conditions. The second issue is how to exploit scoring functions to prune the less relevant elements as early as possible. In this section we discuss these two issues and give an overview for multiple examples of top-k techniques in XML databases.

7.1. The TopX System

It is typical in XML databases that multiple indexes are built for different content and/or structure conditions. Processing multiple indexes to identify the top-k objects is extensively studied by top-k algorithms in relational databases, for example, the TA family of algorithms [Fagin et al. 2001]. This has motivated approaches that extend relational top-k algorithms in XML settings. The TopX system [Theobald et al. 2005] is

one example of such approaches. TopX builds on previous work [Theobald et al. 2004], discussed in Section 5.1.2. The proposed techniques focus on inexpensive sorted accesses to ranked document lists, with scheduled random accesses. For each document, local scores coming from ranked lists are aggregated into global scores based a monotonic score aggregation function such as weighted summation.

Ranked lists in TopX maintain different orders for the documents corpus based on different content and structure conditions. These lists are implemented as relational tables indexed using various B+-tree indexes. The ranked lists are used primarily to evaluate content conditions in block-scan fashion. The evaluation of expensive structure conditions is postponed or avoided by scheduling random accesses only when they are cost-beneficial.

The computational model of TopX is based on the traditional XML element-tree model, where each tree node has a tag and content. The *full content* of a node is defined as the concatenation of the contents of all node's descendants. Different content scoring measures are adopted based on the contents, or the full contents of a node n with tag A:

—Term frequency, $tf(t, n)$, of term t in node n, is the number of occurrences of t in the content of n.

—Full term frequency, $ftf(t, n)$, of term t in node n, is the number of occurrences of t in the full content of n.

—Tag frequency, N_A, of tag A, is the number of nodes with tag A in the entire document corpus.

—Element frequency, $ef_A(t)$, of term t with respect to tag A, is the number of nodes with tag A that contain t in their full contents in the entire document corpus.

For example, consider the content condition "$A//t_1, t_2 \ldots t_m$", where A is a tag name and $t_1 \cdots t_m$ are terms that occur in the *full contents* of A. The score of node n with tag A, with respect to the above condition, is computed as follows:

$$score(n, A//t_1 \ldots t_m) = \frac{\sum_{i=1}^{m} relevance_i.specificity_i}{compactness(n)}, \tag{4}$$

where $relevance_i$ reflects *ftf* values, $specificity_i$ reflects N_A and $ef_A(t)$ values, and $compactness(n)$ considers the subtree or element size for normalization. This content scoring function is based on Okapi BM25 scoring model [Robertson and Walker 1994]. The following example illustrates the processing of TopX to answer top-k queries.

Example 7.1 (*TopX Example*). We consider the example illustrated by Theobald et al. [2004] and depicted by Figure 22, where a sample document set composed of three documents is used for illustration. The numbers beside elements refer to their order in pre-order tree traversal. Consider the following query: //A[.//B[.//"b"] and .//C[.//"c"]]. This query requests elements with tag name A and two child tags B and C containing terms b and c, respectively. Table IV shows the element scores of different content conditions computed as *ftf* scores normalized by the number of terms in a subtree, for simplicity. For instance, the (tag:term) pair $(A : a)$ for element 10 in document d_3 has a score of 1/4 because the term a occurs twice among the eight terms under element 10 subtree.

In Example 7.1, finding the top-k matches is done by opening index scans for the two tag:term conditions ($B : b$, and $C : c$), and block-fetching the best document for each of these two conditions. For ($B : b$), the first three entries of index list 2 that belong to the same document d_1 are fetched in a block-scan based on document IDs. Scanning the indexes proceeds in a round-robin fashion among all indexes. A score

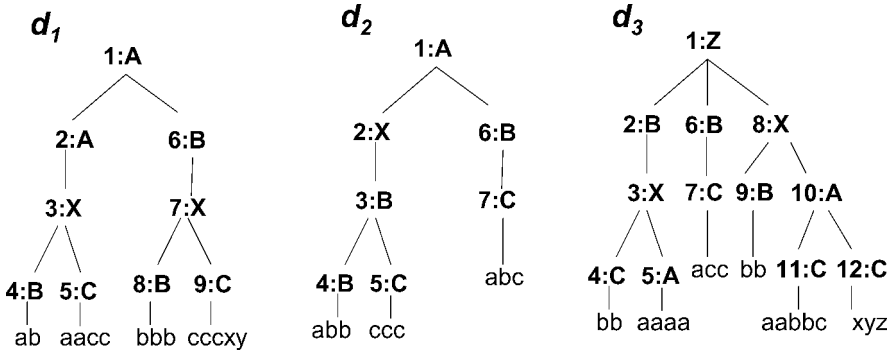

Fig. 22. XML document set.

Table IV. Elements Scores for Different Content Conditions

Index	Tag	Term	Score	DocID	Preorder
1	A	a	1	d3	5
1	A	a	1/4	d3	10
1	A	a	1/2	d1	2
1	A	a	2/9	d2	1
2	B	b	1	d1	8
2	B	b	1/2	d1	4
2	B	b	3/7	d1	6
2	B	b	1	d3	9
2	B	b	1/3	d3	2
2	B	b	2/3	d2	4
2	B	b	1/3	d2	3
2	B	b	1/3	d2	6
3	C	c	1	d2	5
3	C	c	1/3	d2	7
3	C	c	2/3	d3	7
3	C	c	1/5	d3	11
3	C	c	3/5	d1	9
3	C	c	1/2	d1	5

interval [*worstscore*(d), *bestscore*(d)] is computed for each candidate document d, and is updated periodically based on the current candidate scores and the score upper bound of the unseen candidate elements in each list.

Since the first round of block-scan yields two different documents, d_1 and d_2, the second-best document for each condition need to be fetched. After the second round, all d_3's relevant elements for both content conditions are in memory. At this point, a random access for all A elements in d_3 can be triggered, if it is cost-beneficial. This operation efficiently tests the query structure conditions for d_3, that is, whether the $B : b$ and $C : c$ elements are descendants of the same A element. This is done by comparing the preorder and postorder of the respective element pairs. The result is that none of d_3's element satisfies both structure conditions. Notice that the same test cannot be fully performed for document d_1 unless its C tags are fetched first. If *worstscore*(d) is greater than the bottom score of the current top-k set, min_k, then d is inserted into the current top-k set. On the other hand, if *bestscore*(d) is greater than min_k, then d is inserted into a candidate queue and a probabilistic threshold test is periodically performed to examine whether d still has a good chance to qualify for the top-k documents or not.

7.2. The XRank System

As illustrated by Example 7.1, top-k queries in XML databases report the top-k documents (or elements) based on their relevance scores to the given query conditions. A related issue is the notion of keyword proximity, which is the distance between keywords inside documents (elements). Proximity is rather complex to deal with in the hierarchical XML data models because it has to take into account the structure information of elements containing the required keywords. For instance, when keywords are located inside hierarchically distant elements, the proximity should be penalized even if the keywords are physically located near to each other inside the document text. The XRank system [Guo et al. 2003] addresses these issues. XRank considers the problem of producing ranked results for keyword queries over hyperlinked XML documents. The approach of XRank retains the simple keyword search query interface (similar to traditional HTML search engines), while exploiting XML tagged structure during query processing. The adopted scoring function favors more specific results to general results with respect to keyword matching. For instance, the occurrence of all keywords in the same element is favored to their occurrence in distinct elements. The score of an element with respect to one keyword is derived based on its popularity (similar to Google's *PageRank*), and its specificity with respect to the keyword. The element score with respect to all query keywords is the sum of its scores with respect to different keywords, weighted by a measure of keyword proximity.

7.3. XML Structure Scoring

Scoring XML elements based on structure conditions is addressed by Amer-Yahia et al. [2005] with a focus on *twig* queries. A twig query is a rooted tree with string-labeled nodes and two types of edges, / (a child edge) and // (a descendant edge). Three different heuristics of structure relaxation were considered, while scoring XML elements:

—*Edge generalization*. A / edge in query Q is replaced by a // edge to obtain an approximate query $\acute{Q}$.

—*Subtree promotion*. A pattern $a[b[Q1]//Q2]$ is replaced by $a[b[Q1]\ and\ .//Q2]$

—*Leaf node deletion*. A pattern $a[Q1\ and\ .//b]$ where a is the root of the query tree and b is a leaf node is replaced by $a[Q1]$.

These relaxations lead to approximate results that do not necessarily satisfy the original query. Based on these approximations, a directed acyclic graph (DAG), called a *relaxation DAG*, is constructed. The relaxation DAG contains one source node, which is the original query, and one sink node which is the most general relaxed query. An example of relaxation DAG is shown in Figure 23. In this example, the most constraining query is shown in the top node, while its children represent more relaxed queries that are obtained by replacing / constraints by // constraints. Further relaxation is obtained by incrementally removing constraints. The most relaxed query appears at the bottom of the DAG, which only contains the root element.

Elements' scores depend on how close the elements are to the given query. The well-known *tf.idf* measure [Salton and McGill 1983] is adopted to compute elements' scores. The *inverse document frequency* (*idf*), with respect to a query Q, quantifies how many of the elements that satisfy the sink node in the relaxation DAG additionally satisfy Q. The *term frequency* (*tf*) score quantifies the number of distinct ways in which an element matches a query and its relaxations. The *tf* and *idf* scores are used to compute the overall elements' scores. A typical top-k algorithm is used to find the top-k elements based on these scores.

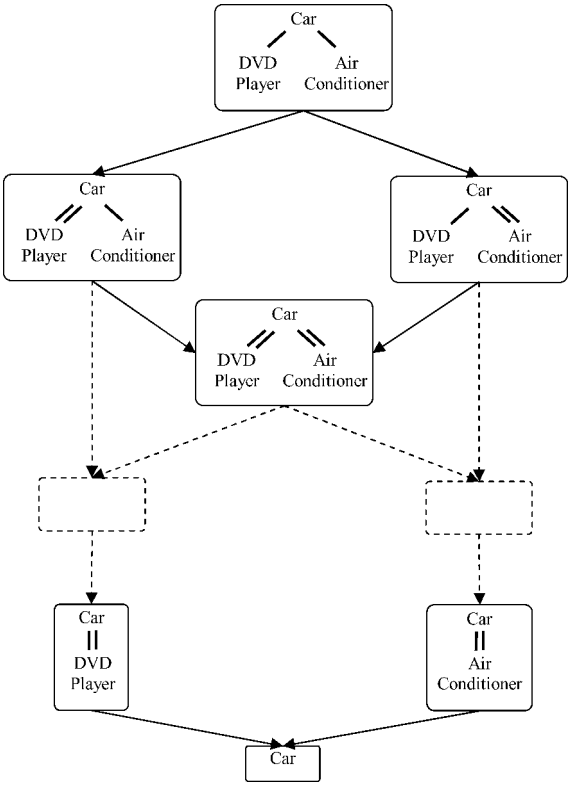

Fig. 23. A query relaxations DAG.

8. VOTING SYSTEMS: THEORETICAL BACKGROUND

In this section, we give a theoretical background for ranking and top-k processing problems in voting theory. The problem of combining different rankings of a list of candidates has deep roots in social and political sciences. Different voting systems have been designed to allow voters to express their preferences regarding a number of candidates and select the winning candidate(s) in a way that satisfies the majority of voters. Voting systems are based on one or a combination of rank aggregation techniques. The underlying rank aggregation methods specify rules that determine how votes are counted and how candidates are ordered. The study of formal voting systems is called *voting theory*.

Choosing a voting system to rank a list of candidates, based on the votes received from several parties, is usually not straightforward. It has been shown that no preference aggregation method could always yield a fair result when combining votes for three or more candidates [Arrow 1951]. The problem with choosing a fair voting system is mainly attributed to the required, but sometimes inevitably violated, properties that a voting system should exhibit [Dwork et al. 2001; Cranor 1996], summarized in the following:

—*Monotonicity.* A voting system is called *monotonic* when raising the valuation for a winning candidate allows it to remain a winner, while lowering the valuation for the candidate lowers its rank in the candidate list. All voting systems that eliminate candidates prior to selecting a winner violate monotonicity.

—*Transitivity.* A voting system is *transitive* if, whenever the rank of x is over the rank of y and y is over z, then it should always be the case that x is ranked over z.

—*Neutrality.* A voting system is *neutral* if it does not favor any specific candidates. Systems that have tie-breaking rules, other than random selection, violate neutrality.

—*Pareto-optimality.* A voting system is *pareto-optimal* if, when every voter prefers candidate x to candidate y, candidate y is not selected.

—*Consistency.* A voting system is *consistent* if dividing the candidate list into two arbitrary parts, and conducting separate voting for each part, results in the same candidate being selected as if the entire candidate list was subjected to voting. If a voting system is consistent, then voting could be conducted in voting domains.

A detailed taxonomy of various voting strategies from the perspectives of social and political sciences can be found in Cranor [1996]. In the following sections, we introduce two of the most widely adopted voting systems. The first system is based on the majority of votes, while the second is based on candidates' relative positions.

8.1. Majority-Based Procedures

Ranking by majority of votes is probably the simplest form of aggregating ranks from different sources. The basic principle is that a group of more than half of the voters should be able to get the outcome they prefer. Ranking by majority for two candidates possesses all desirable properties in voting systems. However, when three or more candidates are under consideration, there may not be a single candidate that is preferred by the majority. In this case, voters compare each pair of candidates in different rounds and another procedure must be used to ensure that the selected candidate is preferred by the majority.

8.1.1. Condorcet Criterion. Ranking candidates through pairwise voting was discovered over 200 years ago by the mathematician and social scientist M. Condorcet in 1785. The criterion was based on a majority rule, but instead of voting for only one candidate, candidates are ranked in order of preference. This can be seen as series of pairwise comparisons between the members of each candidate pair. The end result is a winner who is favored by the majority of voters. Ballots are counted by considering all possible sets of two-candidate elections from all available candidates. That is, each candidate is considered against each and every other candidate. A candidate wins against an opponent in a single ballot if the candidate is ranked higher than its opponent. Whoever has the most votes based on all these one-on-one elections wins.

If a candidate is preferred over all other candidates, that candidate is the Condorcet winner. Condorcet winners may not always exist, due to a fundamental paradox: it is possible for the voters to prefer A over B, B over C, and C over A simultaneously. This is called *majority rule cycle*, and it must be resolved by some other mechanism. A voting system that is able to produce the Condorcet winner is said to attain the *Condorcet property* [Young and Levenglick 1978].

Example 8.1 (*Condorcet Criterion*[2]). This example illustrates the basic counting method of Condorcet criterion. Consider an election for the candidates A, B, C, and D. The election ballot can be represented as a matrix, where the row is the runner under consideration, and the column is the opponent. The cell at (runner, opponent) contains 1 if runner is preferred, and 0 if not. Cells marked "–" are logically zero as a

[2]Available online at the Election Methods Web site `http://www.electionmethods.org`.

	A	B	C	D
A	–	0	0	0
B	1	–	1	1
C	1	0	–	0
D	1	0	1	–

Fig. 24. Voter 1 ballot.

	A	B	C	D
A	–	0	0	0
B	1	–	1	0
C	0	0	–	0
D	1	1	1	–

Fig. 25. Voter 2 ballot.

	A	B	C	D
A	–	0	0	0
B	2	–	2	1
C	1	0	–	0
D	2	1	2	–

Fig. 26. Sum ballot.

candidate can not be defeated by himself. This binary matrix is inversely symmetric: (runner, opponent) is ¬(opponent, runner); however, the sum of all ballot matrices is not symmetric. When the sum matrix is found, the contest between each candidate pair is considered. The number of votes for runner over opponent (runner, opponent) is compared to the number of votes for opponent over runner (opponent, runner). The one-on-one winner has the most votes. If one candidate wins against all other candidates, that candidate wins the election. The sum matrix is the primary piece of data used to resolve majority rule cycles.

Figures 24 and 25 show two sample voter ballots for four candidates A, B, C, and D, while Figure 26 shows the sum ballot. The sum ballot shows that B defeats A with score 2-0, B defeats C with score 2-0, and B and D have equal scores 1-1. In this case, no Condorcet winner could be reached.

An extension of the Condorect criterion has been introduced by Truchon [1998]: if a candidate is consistently ranked ahead of another candidate by an absolute majority of voters, it should be ahead in the final ranking. The term *consistently* refers to the absence of cycles in the majority relation involving these two candidates.

It has been shown that the *extended Condorect criterion* has excellent spam filtering properties when aggregating the rankings generated by different search engines [Dwork et al. 2001]. Specifically, if a spam page is ranked high by fewer than half of the search engines, then the majority of search engines prefer a good page to this spam page. In this case, spam pages will occupy a late rank based on any rank aggregation method that satisfies the *extended Condorect criterion*.

8.1.2. Dodgson Ranking. In 1876, C. Dodgson devised a nonbinary procedure to overcome the problem of voting cycles [Black 1958]. He suggested that one should always

choose the candidate that is "closest" to being a Condorcet winner. His concept of distance was based on the number of inversions of pairs in the individual preferences. A natural extension of Dodgson's method is to rank all candidates with respect to the minimum number of inversions necessary to make each of them a Condorcet winner.

8.1.3. Copeland Rule. Copeland's [1951] method is a Condorcet method in which the winner is determined by finding the candidate with the most pairwise victories. This rule selects the candidate with the largest Copeland index, which is the number of times a candidate beats other candidates minus the number of times that candidate loses to other candidates when the candidates are considered in pairwise comparisons. However, this method often leads to ties when there is no Condorcet winner. For example, if there is a three-candidate majority rule cycle, each candidate will have exactly one loss, and there will be an unresolved tie among the three. The Copeland rule is Pareto-optimal.

The main underlying idea of both Dodgson's and Copeland's methods, namely, the closer a candidate is to be a Condorcet winner the better, is actually identical. However, some anomalies indicate that they can result in different outcomes. Klamler [2003] gave a detailed comparison of the Dodgson ranking and the Copeland rule showing that the Copeland winner can occur at any position in the Dodgson ranking. For some settings of individual preferences of candidates, the Dodgson ranking and the Copeland ranking are exactly the opposite.

8.2. Positional Procedures

Unlike majority-based methods, which are usually based on information from binary comparisons, positional methods take into account the information about voters' preference orderings. These methods do not necessarily select the Condorcet winner when one exists. In addition, for some settings of voters' preferences, each method produces a different outcome. The three positional methods discussed here are all monotonic and Pareto-optimal.

8.2.1. Approval Voting. In this method voters give a single vote to each candidate they approve [Brams and Fishburn 1983]. This is equivalent to giving each candidate a score of $+1$ if the voter approves the candidate, and a score of 0 if the voter does not approve the candidate. Votes are tallied, and the candidates are ranked based on the total number of votes they receive.

8.2.2. Plurality Voting. This method is also referred to as *first–past–the–post* [Nurmi 1987]. The method is based on the *plurality criterion*, which can be simply described as follows: if the number of ballots ranking A as the first preference is greater than the number of ballots on which another candidate B is the first preference, then A's probability of winning must be no less than B's. In other words, plurality voting mainly uses information about each voter's most preferred candidate. When n candidates are to be selected, voters vote for their n most preferred candidates. Because this procedure only takes into account the first preference, or first n preferences, of each voter, it often fails to select the Condorcet winner. As a result, it sometimes produces results which appear illogical when three or more candidates compete. It sometimes even selects the Condorcet loser, the candidate that is defeated by all others in pairwise comparisons.

Example 8.2 (*Plurality Voting*). Imagine an election held to select the winner among four candidates A, B, C, and D. Assume voters have cast their ballots strictly as follows:

42% of voters	26% of voters	15% of voters	17% of voters
1. *A*	1. *B*	1. *C*	1. *D*
2. *B*	2. *C*	2. *D*	2. *C*
3. *C*	3. *D*	3. *B*	3. *B*
4. *D*	4. *A*	4. *A*	4. *A*

Candidate *A* is the winner based on the plurality of votes, even though the majority of voters (58%) did not select *A* as the winner.

8.2.3. Borda Count. Borda count voting , proposed in 1770 by the French mathematician Jean-Charles de Borda, is a procedure in which each voter forms a preference ranking for all candidates. The Borda count is a voting system used for single-winner elections in which each voter orders the candidates. The Borda count is classified as a positional voting system because each rank on the ballot is worth a certain number of points. Given n candidates, each voter orders all the candidates such that the first-place candidate on a ballot receives $n - 1$ points, the second-place candidate receives $n - 2$ points, and in general the candidate in the ith place receives $n - i$ points. The candidate ranked last on the ballot therefore receives zero points. The points are added up across all the ballots, and the candidate with the most points is the winner.

This system is similar to "scoring" methods which incorporate all voter preference information into the vote aggregation. However, as with the other positional voting methods, this does not always produce a logical result. In fact, the result of a Borda count is often a function of the number of candidates considered. However, the Borda count will never choose the Condorcet loser. Borda's method is the basis for most current rank aggregation algorithms.

Example 8.3 (*Borda Count*). In Example 8.2, *B* is the Borda winner in this election, as it has the most points (see the table below). *B* also happens to be the Condorcet winner in this case. While the Borda count does not always select the Condorcet winner as the Borda Count winner, it always ranks the Condorcet winner above the Condorcet loser. No other positional method can guarantee such a relationship.

Candidate	First	Second	Third	Fourth	Total Points
A	42	0	0	58	126
B	26	42	32	0	194
C	15	43	42	0	173
D	17	15	26	42	107

The Borda count is vulnerable to *compromising*. That is, voters can help avoid the election of some candidate by raising the position of a more-preferred candidate on their ballot. The Borda count method can be extended to include tie-breaking methods. Moreover, ballots that do not rank all the candidates can be allowed by giving unranked candidates zero points, or alternatively by assigning the points up to k, where k is the number of candidates ranked on a ballot. For example, a ballot that ranks candidate *A* first and candidate *B* second, leaving everyone else unranked, would give 2 points to *A*, 1 point to *B*, and zero points to all other candidates.

Nanson's method, due to Edward John Nanson (1850–1936), is a procedure for finding the Condorcet winner of a Borda count. In this method, the Borda count is computed for each candidate and the candidate with the lowest Borda count is eliminated, and a new election is held using the Borda count until a single winner emerges. If there is a candidate that is the Condorcet winner, this method chooses that candidate. If there is no Condorcet winner, then some candidate, not necessarily the same as the Borda count winner, is chosen.

8.3. Measuring Distance Between Rankings

An alternative procedure to aggregate the rankings obtained from different sources is using a distance measure to quantify the disagreements among different rankings. An optimal rank aggregation method is the one that induces an overall ranking with minimum distance to the different rankings obtained from different sources. Diaconis [1998] and Fagin et al. [2003, 2004] described different distance measures. We describe in the following two widely adopted measures.

8.3.1. Footrule Distance. The *Footrule*, also called the *Spearman*, *distance* is an absolute distance between two ranked vectors. Given two ranked vectors α and β, for a list of n candidates, the Footrule distance is defined as

$$F(\alpha, \beta) = \sum_{i=1}^{n} |\alpha(i) - \beta(i)|, \tag{5}$$

where $\alpha(i)$ and $\beta(i)$ denote the position of candidate i in α and β, respectively. The maximum value of $F(\alpha, \beta)$ is $\frac{n^2}{2}$ when n is even, and $\frac{(n+1)(n-1)}{2}$ when n is odd.

For example, consider a list of three candidates, $\{A, B, C\}$; if two voters, α and β, order candidates as $\alpha = \{A, B, C\}$ and $\beta = \{C, B, A\}$, then the Footrule distance between α and β is $|1-3|+|2-2|+|3-1| = 4$. The computed distance is the maximum possible distance for lists of three candidates, since one vector ranks the candidates in the reverse order of the other. The Footrule distance can be computed in linear time.

8.3.2. Kendall Tau Distance. The Kendall tau distance measures the amount of disagreement between two rankings by counting the number of pairwise disagreements [Kendall 1945]. Consider two rankings α and β, of a list of n candidates; the Kendall tau distance is defined as

$$K(\alpha, \beta) = |\{(i, j) | i < j,\ \alpha(i) < \alpha(j)\ \textit{while}\ \beta(i) > \beta(j)\}|. \tag{6}$$

The maximum value of $K(\alpha, \beta)$ is $\frac{n(n-1)}{2}$, which occurs if α is the reverse of β. Kendall tau distance can be computed by finding the minimum number of pairwise swaps of adjacent elements needed to convert one ranking to the other. For example, if two voters, α and β, order candidates as $\alpha = \{A, B, C\}$ and $\beta = \{B, C, A\}$, the Kendall tau distance between the two rankings is 2 since it takes two swaps to convert one of the rankings to the other.

Diaconis and Graham [1977] showed that the Kendall tau distance can be approximated using the Footrule distance as $K(\alpha, \beta) \leq F(\alpha, \beta) \leq 2K(\alpha, \beta)$, which shows that Footrule and Kendall tau distance measures are equivalent to each other to some extent.

8.3.3. Kemeny Proposal. The optimal aggregation of different rankings produces a overall ranking with minimum distance with respect to the given rankings. Given n different rankings $\alpha_1, \alpha_2, \ldots, \alpha_n$ for a list of candidates, the normalized Kendall tau distance between the aggregated overall ranking ρ and the given rankings is defined as

$$\textit{Distance}(\rho, \alpha_1, \alpha_2, \ldots \alpha_n) = \sum_{i=1}^{n} (K(\rho, \alpha(i)))/n. \tag{7}$$

Kemeny's proposal has been adopted for performing rank aggregation, based on the Kendall tau distance, in Dwork et al. [2001]. It was shown that computing Kemeny

optimal aggregation is NP-hard even for $n = 4$. Kemeny aggregation satisfies neutrality and consistency properties of voting methods.

The hardness of solving the problem of distance-based rank aggregation is related to the chosen distance measure. In some settings, measures could be adopted to solve the problem in polynomial time, such as measuring the distance between only the top-k lists rather than fully ranked lists. Applying the traditional measures in this case is not possible since it requires accessing the full lists. To work around this problem, extended versions of Kendall tau and Footrule distance measures were adopted by Fagin et al. [2003]. The main idea is to truncate the top-k lists at various points $i \leq k$, compute the symmetric difference metric between the resulting top i lists, and take a suitable combination of them. Different cases are considered such as an item pair that appears in all top-k lists as opposed to one or both items missing from one of the lists. A penalty is assigned to each case while computing the distance measures.

9. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

We have surveyed top-k processing techniques in relational databases. We provided a classification for top-k techniques based on several dimensions such as the adopted query model, data access, implementation level, and supported ranking functions. We discussed the details of several algorithms to illustrate the different challenges and data management problems they address. We also discussed related top-k processing techniques in the XML domain, as well as methods used for scoring XML elements. Finally, we provided a theoretical background of the ranking and top-k processing problems from voting theory.

We envision the following research directions to be important to pursue:

—*Dealing with uncertainty.* Efficient processing of top-k queries that deal with different sources of uncertainty and fuzziness, in both data and queries, is a challenging task. Designing uncertainty models to meet the needs of practical applications, as well as extending relational processing to conform with different probabilistic models, are two important issues with many unsolved problems. Exploiting the semantics of top-k queries to identify optimization chances in these settings is another important question.

—*Cost models.* Building generic cost models for top-k queries with different ranking functions is still in its primitive stages. Leveraging the concept of rank-awareness in query optimizers, and making use of rank-aware cost models is an important related direction.

—*Learning ranking functions.* Learning ranking functions from users' profiles or feedback is an interesting research direction that involves many practical applications, especially in Web environments. Building intelligent systems that recognize user's preferences by interaction, and optimizing data storage and retrieval for efficient query processing, are two important problems.

—*Privacy and anonymization.* Most current top-k processing techniques assume that the rankings obtained from different sources are readily available. In some settings, revealing such rankings might be restricted or anonymized to protect private data. Processing top-k queries using partially disclosed data is an interesting research topic.

REFERENCES

ACHARYA, S., GIBBONS, P. B., AND POOSALA, V. 2000. Congressional samples for approximate answering of group-by queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. 487–498.

AMATO, G., RABITTI, F., SAVINO, P., AND ZEZULA, P. 2003. Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inform. Syst. 21,* 2, 192–227.

AMER-YAHIA, S., KOUDAS, N., MARIAN, A., SRIVASTAVA, D., AND TOMAN, D. 2005. Structure and content scoring for xml. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 361–372.

AREF, W. G., CATLIN, A. C., ELMAGARMID, A. K., FAN, J., HAMMAD, M. A., ILYAS, I. F., MARZOUK, M., PRABHAKAR, S., AND ZHU, X. 2004. VDBMS: A testbed facility for research in video database benchmarking. *ACM Multimed. Syst. J.* (Special Issue on Multimedia Document Management Systems) *9*, 6, 575–585.

ARROW, K. 1951. *Social Choice and Individual Values*. Wiley, New York, NY.

BABCOCK, B., CHAUDHURI, S., AND DAS, G. 2003. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 539–550.

BARBARÁ, D., GARCIA-MOLINA, H., AND PORTER, D. 1992. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng. 4,* 5, 487–502.

BLACK, D. 1958. *The Theory of Committees and Elections*. Cambridge University Press, London, U.K.

BÖRZSÖNYI, S., KOSSMANN, D., AND STOCKER, K. 2001. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*. 421.

BRAMS, S. J. AND FISHBURN, P. C. 1983. *Approval Voting*. Birkhauser, Boston, MA.

BRUNO, N., CHAUDHURI, S., AND GRAVANO, L. 2002a. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst. 27,* 2, 153–187.

BRUNO, N., GRAVANO, L., AND MARIAN, A. 2002b. Evaluating top-*k* queries over Web-accessible databases. In *Proceedings of the 18th International Conference on Data Engineering*. 369.

CHAKRABARTI, K., GAROFALAKIS, M., RASTOGI, R., AND SHIM, K. 2001. Approximate query processing using wavelets. *VLDB J. 10,* 2–3, 199–223.

CHANG, K. C. AND HWANG, S. 2002. Minimal probing: supporting expensive predicates for top-*k* queries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 346–357.

CHANG, Y., BERGMAN, L. D., CASTELLI, V., LI, C., LO, M., AND SMITH, J. R. 2000. The Onion Technique: Indexing for linear optimization queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. 391–402.

CHAUDHURI, S., DAS, G., DATAR, M., MOTWANI, R., AND NARASAYYA, V. R. 2001a. Overcoming limitations of sampling for aggregation queries. In *Proceedings of the 17th International Conference on Data Engineering*. 534–542.

CHAUDHURI, S., DAS, G., AND NARASAYYA, V. 2001b. A robust, optimization-based approach for approximate answering of aggregate queries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 295–306.

CHAUDHURI, S., MOTWANI, R., AND NARASAYYA, V. 1998. Random sampling for histogram construction: How much is enough? In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. 436–447.

COPELAND, A. H. 1951. A reasonable social welfare function. Mimeo. University of Michigan, Ann Arbor, MI.

CRANOR, L. 1996. Declared-strategy voting: An instrument for group decision-making. Ph.D. dissertation. Washington University, St. Louis, MO.

DAS, G., GUNOPULOS, D., KOUDAS, N., AND TSIROGIANNIS, D. 2006. Answering top-*k* queries using views. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 451–462.

DIACONIS, P. 1998. *Group Representation in Probability and Statistics*. Institute of Mathematical Statistics. Web site: `www.imstat.org`.

DIACONIS, P. AND GRAHAM, R. 1977. Spearman's footrule as a measure of disarray. *J. Roy. Statist. Soc. Series B 39,* 2, 262–268.

DONJERKOVIC, D. AND RAMAKRISHNAN, R. 1999. Probabilistic optimization of top *N* queries. In *Proceedings of the 25th International Conference on Very Large Data Bases*. 411–422.

DWORK, C., KUMAR, S. R., NAOR, M., AND SIVAKUMAR, D. 2001. Rank aggregation methods for the Web. In *Proceedings of the 10th International Conference on World Wide Web*. 613–622.

FAGIN, R., KUMAR, R., MAHDIAN, M., SIVAKUMAR, D., AND VEE, E. 2004. Comparing and aggregating rankings with ties. In *Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 47–58.

FAGIN, R., KUMAR, R., AND SIVAKUMAR, D. 2003. Comparing top *k* lists. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. 28–36.

FAGIN, R., LOTEM, A., AND NAOR, M. 2001. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci. 1,* 1, 614–656.

FUHR, N. 1990. A probabilistic framework for vague queries and imprecise information in databases. In *Proceedings of the 16th International Conference on Very Large Data Bases*. 696–707.

GANTI, V., LEE, M., AND RAMAKRISHNAN, R. 2000. ICICLES: Self-tuning samples for approximate query answering. In *Proceedings of the 26th International Conference on Very Large Data Bases*. 176–187.

GETOOR, L. AND DIEHL, C. P. 2005. Link mining: A survey. *ACM SIGKDD Explor. Newsl. 7,* 2, 3–12.

GÜNTZER, U., BALKE, W., AND KIEßLING, W. 2000. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*. 419–428.

GÜNTZER, U., BALKE, W., AND KIEßLING, W. 2001. Towards efficient multi-feature queries in heterogeneous environments. In *Proceedings of the International Conference on Information Technology: Coding and Computing*. 622.

GUO, L., SHAO, F., BOTEV, C., AND SHANMUGASUNDARAM, J. 2003. XRANK: Ranked keyword search over XML documents. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 16–27.

HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. 1997. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. 171–182.

HOEFFDING, W. 1963. Probability inequalities for sums of bounded random variables. *American Statistical Association Journal*, 13–30.

HRISTIDIS, V., KOUDAS, N., AND PAPAKONSTANTINOU, Y. 2001. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 259–270.

HRISTIDIS, V. AND PAPAKONSTANTINOU, Y. 2004. Algorithms and applications for answering ranked queries using ranked views. *VLDB J. 13,* 1, 49–70.

HWANG, S. AND CHANG, K. C. 2007a. Optimizing top-*k* queries for middleware access: A unified cost-based approach. *ACM Trans. Database Syst. 32,* 1, 5.

HWANG, S. AND CHANG, K. C. 2007b. Probe minimization by schedule optimization: Supporting top-*k* queries with expensive predicates. *IEEE Trans. Knowl. Data Eng. 19,* 5, 646–662.

ILYAS, I. F., AREF, W. G., AND ELMAGARMID, A. K. 2002. Joining ranked inputs in practice. In *Proceedings of the 28th International Conference on Very Large Data Bases*. 950–961.

ILYAS, I. F., AREF, W. G., AND ELMAGARMID, A. K. 2004a. Supporting top-*k* join queries in relational databases. *VLDB J. 13,* 3, 207–221.

ILYAS, I. F., AREF, W. G., ELMAGARMID, A. K., ELMONGUI, H. G., SHAH, R., AND VITTER, J. S. 2006. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst. 31,* 4, 1257–1304.

ILYAS, I. F., SHAH, R., AREF, W. G., VITTER, J. S., AND ELMAGARMID, A. K. 2004b. Rank-aware query optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 203–214.

ILYAS, F. I., WALID, G. A., and ELMAGARMID, A. K. 2003. Supporting top-*k* join queries in relational databases. In *Proceedings of the 29th International Conference on Very Large Databases.* 754–765.

IMIELINSKI, T. AND LIPSKI JR., W. 1984. Incomplete information in relational databases. *J. ACM 31,* 4, 761–791.

KENDALL, M. G. 1945. The treatment of ties in ranking problems. *Biometrika 33,* 3, 239–251.

KLAMLER, C. 2003. A comparison of the dodgson method and the copeland rule. *Econ. Bull. 4,* 8, 1–7.

LI, C., CHANG, K. C., AND ILYAS, I. F. 2006. Supporting ad-hoc ranking aggregates. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. 61–72.

LI, C., CHANG, K. C., ILYAS, I. F., AND SONG, S. 2005. RankSQL: Query algebra and optimization for relational top-*k* queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. 131–142.

MAMOULIS, N., CHENG, K. H., YIU, M. L., AND CHEUNG, D. W. 2006. Efficient aggregation of ranked inputs. In *Proceedings of the 22rd International Conference on Data Engineering*. 72.

MARIAN, A., BRUNO, N., AND GRAVANO, L. 2004. Evaluating top-*k* queries over Web-accessible databases. *ACM Trans. Database Syst. 29,* 2, 319–362.

MICHEL, S., TRIANTAFILLOU, P., AND WEIKUM, G. 2005. KLEE: A framework for distributed top-*k* query algorithms. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 637–648.

NATSEV, A., CHANG, Y., SMITH, J. R., LI, C., AND VITTER, J. S. 2001. Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th International Conference on Very Large Data Bases*. 281–290.

NURMI, H. 1987. *Comparing Voting Systems*. D. Reidel Publishing Company, Dordrecht, Germany.

RÉ, C., DALVI, N. N., AND SUCIU, D. 2007. Efficient top-k query evaluation on probabilistic data. In *Proceedings of the 23rd International Conference on Data Engineering*. 886–895.

ROBERTSON, S. E. AND WALKER, S. 1994. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 232–241.

SALTON, G. AND MCGILL, M. J. 1983. *Introduction to Modern IR*. McGraw-Hill, New York, NY.

SILBERSTEIN, A., BRAYNARD, R., ELLIS, C. S., MUNAGALA, K., AND YANG, J. 2006. A sampling-based approach to optimizing top-k queries in sensor networks. In *Proceedings of the 22nd International Conference on Data Engineering*. 68.

SOLIMAN, M. A., ILYAS, I. F., AND CHANG, K. C.-C. 2007. Top-k query processing in uncertain databases. In *Proceedings of the 23rd International Conference on Data Engineering*. 896–905.

THEOBALD, M., SCHENKEL, R., AND WEIKUM, G. 2005. An efficient and versatile query engine for TopX search. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 625–636.

THEOBALD, M., WEIKUM, G., AND SCHENKEL, R. 2004. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the 30th International Conference on Very Large Data Bases*. 648–659.

TRUCHON, M. 1998. An extension of the Condorcet criterion and Kemeny orders. Cahier 98-15. Centre de Recherche en Economie et Finance Appliquees, Université Laval, Québec, Canada.

TSAPARAS, P., PALPANAS, T., KOTIDIS, Y., KOUDAS, N., AND SRIVASTAVA, D. 2003. Ranked join indices. In *Proceedings of the 19th International Conference on Data Engineering*. 277.

VRBSKY, S. V. AND LIU, J. W. S. 1993. APPROXIMATE—a query processor that produces monotonically improving approximate answers. *IEEE Trans. Knowl. Data Eng. 5,* 6, 1056–1068.

XIN, D., CHEN, C., AND HAN, J. 2006. Towards robust indexing for ranked queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 235–246.

XIN, D., HAN, J., AND CHANG, K. C. 2007. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. 103–114.

YOUNG, H. P. AND LEVENGLICK, A. 1978. A consistent extension of condorcet's election principle. *SIAM J. Appl. Math. 35,* 2, 285–300.

YUAN, Y., LIN, X., LIU, Q., WANG, W., YU, J. X., AND ZHANG, Q. 2005. Efficient computation of the skyline cube. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 241–252.

ZHANG, Z., HWANG, S., CHANG, K. C., WANG, M., LANG, C. A., AND CHANG, Y. 2006. Boolean + ranking: Querying a database by k-constrained optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. 359–370.

Received January 2007; revised July 2007; accepted November 2007