Universität Augsburg

On Real-time Top *k* Querying for Mobile Services

Wolf-Tilo Balke, Ulrich Güntzer, Werner Kießling

Report 2002-2 Februar 2002

Institut für Informatik
D – 86135 Augsburg

On Real-time Top *k* Querying for Mobile Services

Wolf-Tilo Balke[*] **Ulrich Güntzer**[+] **Werner Kießling**[*]

[*]Institute of Computer Science
University of Augsburg
Augsburg, Germany
{balke, kiessling}@informatik.uni-augsburg.de

[+]Institute of Computer Science
University of Tübingen
Tübingen, Germany
guentzer@informatik.uni-tuebingen.de

Abstract

Mobile services offering multi-feature query capabilities must meet tough response time requirements to gain customer acceptance. The top-k query model is a popular candidate to implement such services. Focusing on a central server architecture we present a new algorithm called SR-Combine that closely self-adapts itself to particular cost ratios in such environments. SR-Combine optimizes both the object accesses and the query run-times. We perform a series of synthetical benchmarks to verify the superiority of SR-Combine over existing algorithms. In order to assess whether it can meet the stated response time requirements for mobile access, we propose a psychologically founded model. It turns out that for a wide range of practical cases SR-Combine can satisfy these goals. Where this isn't yet the case, we show up ways how to get there systematically. Thus with SR-Combine a breakthrough in real-time capabilities of top-k querying for mobile services is in sight now.

1. Introduction

Top *k* querying has proven to be an increasingly demanding problem in today's applications. Areas like web-based information services, enterprise information systems or content-based media retrieval already make intensive use of this new paradigm. A variety of applications in relational databases [CK97], multi-media databases [OR+98] or recent approaches in location-based services [DGS00] also show that top *k* queries are essential for advanced database retrieval.

Example 1: Business Document Repositories
Consider the case of an insurance company storing their letters and e-mails, contracts and related documents on a central server. At each damage event the person in charge wants to know about recent similar cases or the same persons involved in different cases. Thus a query has to be performed like: "Give me the top cases preferably recently with most similar kinds and costs of damages and same people involved."

Even in this typical example we can already identify four classifiers (kind, costs, people, time), assign scores and sort documents into lists ordered by their relevance towards these issues (what will be called a *stream* in the following). However, when it comes to real-time capabilities for combining streams, current approaches show their limitations. Psychology [Pop97] teaches that users only tend to accept response times up to 3 seconds before their questions are answered. This real-time restriction can generally be applied to on-line search engines and users will allow higher run-times only for very difficult tasks (e.g. in work environments), where they strongly depend on the results. In mobile applications –especially if used in private life– users of course don't want to wait long, but apply strict response-time expectations. In this area there are even monetary reasons involved, because the connection (and the service provided) is often charged with respect to the usage time.

Recent algorithms for top *k* querying [Fag99, GBK00, GBK01, FLN01, BGM02] served only special applications and tended to deteriorate in efficiency very quickly, if the application environment or score distributions for various queries changed. Besides, no real-time results for algorithms in different environments have been published so far (except [BGM02] who reported hours for distributed mobile services). This is because –though optimizing the total number of object accesses– runtimes will nevertheless considerably increase, if the algorithms extensively use expensive kinds of object accesses, e.g. [WH+99]. Thus not only the overall number, but also the different costs for different kinds of accesses and the specific application architecture have to be taken into account. And what is more, when discussing real-time capabilities, not only the cost of database accesses, but also CPU costs have to be considered to build competitive algorithms.

For instance the naïve approach of computing the overall scores for all the objects in the collection obviously hardly causes any CPU costs, but uses lots of database accesses. This trade-off is especially important for the use in environments for mobile services, where small and medium databases ($N \approx 10000$) prevail and bandwidth is low. Thus the setting for new algorithms is even harder than traditional top k querying, where CPU costs could be ignored for large database sizes and only the number of object accesses counted. Due to the field of application we will have to focus on few streams to combine ($n < 10$), due to the technical constraints of mobile devices on small numbers of return values ($k = 3$ to 10) and due to our real-time requirements on response times of only a few seconds.

Our paper is organized as follows. We will take a look on mobile services, several application scenarios and the recently proposed top k querying in section 2. The algorithm *SR-Combine* will be presented in section 3. In section 4 we will focus on efficiency tests and present real-time benchmarks for *SR-Combine* in different environments. Finally will close with a short summary and outlook.

2. Top *k* Querying for Mobile Services

2.1 Characteristics of Mobile Services

With the current developments of devices like cell phones or PDAs, mobile services will play an important role in future information technology. Pervasive access on information becomes more and more attractive not only for work, but also for private uses. We will illustrate this by three scenarios of different mobile services and discuss their specific characteristics and requirements.

Scenario 1
Mobile Access to global stock exchanges. Typical queries are like: "Get me the five top performers in the area of IT industries that are involved in bio-technology." The querying for this type of service is mainly attribute-based and in general there is virtually no difference in the costs of sorted accesses and random accesses (like in [NR99]) (cf. sec. 2.2).

Scenario 2
Location-based restaurant service. Typical queries are like: "Get me the five nearest restaurants with Asian cuisine, top category and prices of about 40$ per meal." The querying is a mixture of attribute-based queries and spatial retrieval. Random accesses are generally much more expensive than sorted accesses ([BGM02] determined a factor of 10 for this kind of service).

Scenario 3
Mobile on-line auctions. Typical queries are like: "Get me the top five impressionist paintings, in rather brown and green colors that have the lowest prices." The querying involves attribute-based parts often together with multimedia attributes or extracted features [WBK01]. Random accesses in this case are more expensive than sorted accesses. Our practical tests show factors of about 6 (cf. section 4.3).

service	complexity	updates	sources
stock market	low	often	external
location-based	medium	seldom	mixed
mobile auctions	high	seldom	central

As shown in the table above the three scenarios not only differ in the costs for object accesses, but also in characteristics like update behavior and use of external sources. Whereas attribute-based stock market information has to be updated in ranges of few seconds, the information for more complex services involving e.g. multimedia data, is somewhat more durable. Typical update intervals in these cases range between days and weeks. This means that e.g. stock market information has to be imported directly from content providers. But the service provider can transfer durable information like location-based or multimedia data on central servers enabling efficient storage and indexing schemes.

The research area of data integration over the web [GB97, TSH01, GW00] has lead to several architectures for different applications. Architectures for mobile services are mainly twofold:

- **Central Server Architecture (CSA):** If the service provider is also the content provider or handles mainly durable information, services are provided using a high performance server with central data repositories.
- **Distributed Sources Architecture (DSA):** If the service provider and content provider are different or short update ranges are necessary, services may be provided using an application server gathering information on demand from distributed external data sources accessed via the Internet.

Enabling top k queries in mobile environments, however, poses some severe problems depending on the middleware capabilities of algorithms. In [BGM02] ways to build mobile services with a DSA architecture providing direct access to various Internet sources are presented. Though some of these services (e.g. location based city maps or restaurant guides, etc.) would be desirable, tests in [BGM02] also show that the processing times for such simple tasks often need hours.

When trying to meet real-time requirements, we have to build all three scenarios on a CSA architecture. Thus first of all we have to address the problem of combining

Internet sources with local database servers. A solution is given by [GW00] who describe the WSQ/DSQ approach that handles direct accesses to Internet sources in an asynchronous manner and caches the results for later use in virtual tables of a central database server. Since the service provider obviously knows what type of queries to expect, what data is commonly accessed and how often updates are needed to meet the service design, a caching strategy with asynchronous updates is suitable for mobile services. Figure 1 outlines the intended architecture.

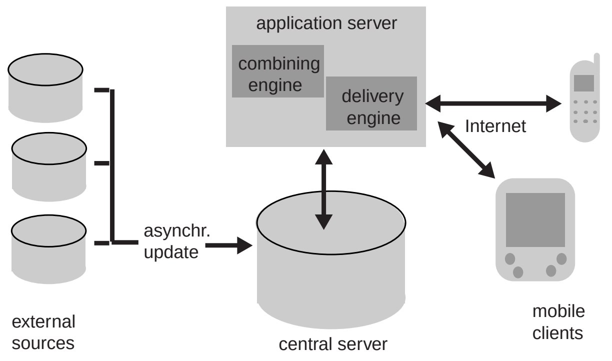

Fig 1: Intended CSA architecture

The mobile service consists of a application server (or a proxy like in [GB97]) containing a combining engine which runs the *SR-Combine* algorithm. All the content is retrieved from a central server (that may be updated in an asynchronous manner). As shown in e.g. [WBK01] with the example of mobile online auctions the delivery engine can transform e.g. generic XML formats using XSLT to support any mobile device e.g. via WAP or i-mode gateways. Another advantage of this architecture is that all data on the local server can be indexed to suit the design of the service. Through statistical analysis also the costs for certain usage patterns can be estimated providing the cost estimations to determine the ratio between different kinds of access personalized for each user. Besides, due to this architecture for all objects different kinds of access are possible, what posed a problem in [BGM02].

2.2 Top *k* Querying Revisited

Since top *k* querying is an important feature not only of mobile services, we will revisit some recently proposed algorithms that can later be used as a yard stick. There have been some different approaches in literature to solve the problem of top k querying. In particular they can be divided into two different categories. Those guaranteeing a correct result set and those using statistical data to get the result set more efficiently [CG99, DR99], however with a specific amount of uncertainty about the correctness of the retrieval result. Due to the nature of mobile services with restricted end devices (bandwidth, display-size, etc.) demanding rather small numbers of objects to be output, in the following we will only refer to those delivering a correct result set to improve chances, that many of the few objects returned are relevant, i.e. we need high precision.

Most applications of top *k* querying algorithms are mainly concerned with gathering information from a variety of data sources. Generally speaking all known algorithms always rely on two kind of access methods providing basic scores:

- The **sorted access** (SA) ranks database objects according to their score values (descending) with respect to a single feature and accesses objects rank by rank.
- The **random access** (RA) can be posed to a data source retrieving the score value of a single object with respect to a single stream.

[BGM02] addresses the problem of sources that may be restricted to either handle sorted accesses (so called S-sources) or random accesses (so-called R-sources). However, due to our CSA architecture we can always rely on both kinds of accesses (so-called SR-sources) and thus our algorithm presented here is called *SR-Combine*. In the following we will always rely on SR-sources.

Having gathered the basic score values by sorted or random access, the total score is determined using a suitable monotonic combining function *F*. Both access methods can however essentially differ in their respective costs. As a rule of thumb it can be stated that the more heterogeneous the environment gets, the more expensive the random access will be compared to sorted accesses. This leads to a large variety of environments ranging from those where random accesses are cheaper than sorted accesses [NR99], via those where random accesses are more expensive with a certain factor [WH+99], to those where random accesses are virtually impossible [Coh98].

Generally speaking algorithms for top *k* querying try to minimize the number of database objects that have to be accessed before being able to deliver a correct result set. The first algorithm in this area is given by Fagin's algorithm [Fag96]. Early approaches by [NR99] for applications in fuzzy logic and by [OR+98] adapted for use in multimedia content-based retrieval, however, presented a threshold algorithm that performed essentially better. This threshold algorithm was generalized to any monotonic combining function and improved with a heuristic control flow in the Quick-Combine approach. A detailed analysis together with first performance results of Quick-Combine is given in [GBK00]. The basic Quick-Combine has been proven to be optimal in minimizing the necessary sorted accesses for top *k* querying [FLN01]. However, for each sorted access Quick-Combine has to perform (*n*-1) random accesses to guarantee the earliest possible termination. Since costs for random accesses may explode in some environments, minimizing the overall object accesses may not always be the best strategy. In some environments it can be far cheaper to replace some random accesses by a larger number of sorted accesses.

This has lead to the development of algorithms like Stream-Combine or NRA [GKB01, FLN01] that do not need any random accesses at all. However, in practical test algorithms of this type quickly showed their limited applicability by accessing 80-90% of all database objects in environments, where skewed data is involved, thus quickly loosing their speed-up even over the naïve approach. Extensive performance tests in [GBK00] show that the Quick-Combine achieves its best results, if skewed data is involved, whereas the Stream-Combine approach performs best in environments with uniformly distributed data. For a later benchmark we will investigate this behavior a little closer in section 4.2.

To overcome the limitations of Stream-Combine a new algorithm called CA (combined algorithm) is given in [FLN01]. CA runs like Stream-Combine, but periodically performs random accesses like Quick-Combine on most promising objects. However, it shows the same characteristic behavior and limitations as the Quick-Combine approach. Thus a mere combination of known algorithms will not solve our problem, but we have to design a new paradigm to get a competitive algorithm that self-adapts to different environments.

3. The *SR-Combine* Algorithm

In the following we will present the *SR-Combine* algorithm overcoming the disadvantages of previous algorithms by suitably self-adapting to a variety of environments and controlling run-time costs. The algorithm *SR-Combine* consists of three phases that successively retrieve k overall best objects. To understand the algorithm more easily we will first present a sketch of the different phases with k=1 and then present some heuristics that lead to a more efficient implementation of our algorithm.

Algorithm *SR-Combine*

- **Phase 1 (Pruning Phase):**
  Phase 1 gathers objects from the different streams until it can be guaranteed that at least one overall best object has been seen among those objects gathered. It can use two techniques: Sorted accesses are used to see scores of (new) objects in streams in descending order and random accesses are used to complete the scores for objects already seen. For each object a lower and an upper bound estimation for the object's total score can be made and there is at least one object that definitely has a better or equal overall score than all objects not seen so far.
- **Phase 2 (Identifying Barriers):**
  Phase 2 divides all objects seen in phase 1 into those that have no chance of being the top object and those that have a chance, the so-called barrier objects. After phase 2 we know that the overall best object is among the barrier objects and all the other objects can be excluded in our search for the overall best object.
- **Phase 3 (Removing Barriers):**
  Phase 3 successively completes scores of barrier objects to get more information about each object. Again both sorted and random accesses are used depending on the cost-ratio. Whenever the information about any object is sufficient to exclude it from the barrier objects, the object can be removed and phase 3 focuses on the remaining barrier objects. If enough barrier objects have been removed, the algorithm can output the overall best object and start over to get the next overall best object.

In general the three phases of this algorithm are quite common in the top k querying literature. In fact most algorithms can be brought down to these three phases, however using different approaches to fulfill each phase. Thus in the following we will present our retrieval model, have a look at each phase and then present the full algorithm.

3.1 A Faster (Top k)* Retrieval Model

Since the correctness of the result set returned strongly depends on the correct scoring of the underlying retrieval system(s), the exact order of the best k matches can be neglected during the retrieval phases. As the system successively outputs objects, any object can already be returned, if we can be sure that it will definitely belong to the set of k best matches, no matter what its exact rank or score in the final result set will be. Thus, outputting an object our (top k)* retrieval model states that it belongs to the k best objects, but does not yet determine whether it will be the first ranked or the k-th ranked object (of course, after the whole set of k objects has been output, the set can easily be ordered). Though tests have shown, that this retrieval model will need the same total time to deliver all k results, it essentially improves the time needed before some first objects can be output. This is not only important for psychological reasons, but gives the user an earlier idea of the result set objects. While the algorithm is still running, the user thus can use the meantime to decide, if the results are already satisfying and the algorithm can be terminated early or if the query itself has to be improved, etc.

In our case the advantage of this retrieval model is the assertion that we can already safely return the first object, if there are only (k-1) barrier objects left. Obviously, this is because if only (k-1) objects have a chance of being more relevant than our object, it may be the overall best object, if all barriers can be removed, but it will at least the k-th best object, even if all (k-1) barriers emerge to have higher scores. Inductively we can conclude, that if a second object should be output, there may at most be (k-2) barriers, for the third object (k-3) barriers, and so on. On the other hand this leads to the observation, that when removing barriers for the output of a first object, we may focus on a set of k objects e.g. those having the highest upper bound estimations and only need to have a look at the remaining barriers whenever one of our k barriers is removed. Inductively this again leads to a restriction to a set of (k - (number of objects returned)) objects, which have to be updated regularly during the removing barriers phase.

3.2 The Pruning Phase

To get an efficient pruning phase we will now state a few heuristics. Some of these will be helpful in the third phase of removing barriers also.

Heuristic 1: Taking the Environment into Account
Previous work shows that the costs for sorted and random accesses may strongly differ depending on the environment. Very heterogeneous environments like in [WH+99, Coh98] show very high costs for random accesses, in contrast environments like in [NR99] allow random accesses that are even cheaper than sorted accesses. Thus strategies like optimizing the total number of accesses, however using an expensive kind of access may fail. Thus the algorithm's runtime may explode, if the specific cost-ratio between sorted and random accesses is not taken into account.

Since some sorted accesses are necessary to see new objects (if "wild guesses" are prohibited, which was shown to be reasonable in [FLN01]), during the entire algorithm our control flow takes care, that –based on the cost-ratio– for each sorted access only as many random accesses are performed, that the total runtime costs may at most be doubled, though the total number of objects accesses is optimized. Thus the trade-off between minimizing total object accesses and exploding the runtime by using more cost expensive accesses is bounded. This technique is not only used within the pruning phase, but can also be adapted to the removing barriers phase. ■

Heuristic 2: Using the Data Distribution
The data distribution in each stream may severely differ. There may be streams with quickly decreasing scores (especially if skewed data is involved), streams with uniformly distributed score values or streams that provide very similar or even the same score for a large number of objects (e.g. if exact matches are involved and all the database objects are split in perfect matches (1.0) or no match (0.0)). Of course the evaluation of quickly decreasing streams should be preferred, because in contrast to constant streams, they discriminate well between objects. A second important factor to estimate a certain stream's influence on the final result is the stream's weight that is used in the combining function. Obviously highly weighted streams should be preferred.

In [GBK00] an indicator technique is proposed that uses the relative decrease of a stream's score distribution together with a partial derivative of the combining function (e.g. a weighted mean) for the stream (if the derivative exists, otherwise - e.g. in the case of max as combining function - this part is set to $1/n$). For the relative decrease in each stream we use the simple difference between the score of the last seen object (o_{last}) and the score of the p-th last seen object ($o_{p\text{-}th\ last}$), where p can be any natural number. Smaller values of p estimate the local behavior of a stream, larger values its more global behavior. In practical tests the choice of p=5 has lead to the best results, thus we will use p=5 in all our later tests. Our indicators for each stream are given by:

$$\Delta_j := \left| \partial F / \partial x_j \right| \cdot (s_j(o_{last}) - s_j(o_{p\text{-}th\ last})) \quad (1 \le j \le n)$$

Always choosing the right stream for sorted accesses can lead to an important improvement factor for the algorithm's real time capabilities. Theoretically at most a factor of n can be reached over a classical round robin strategy. Since practical tests show that even in the case of only three streams involved our simple indicator gains a factor of 2 (in run-times and object accesses, see diagram below) virtually without causing computing costs, we adapted this indicator for the use in our new algorithm and thus halved its runtime. Of course after each sorted access performed on a stream the indicator for this stream has to be recalculated. Please note, that we inserted a line in all our real-time diagrams that marks the 3 seconds requirement. As can be seen in fig. 2 our Indicator technique helps to gain real-time capabilities (section 4 will extensively deal with performance issues).

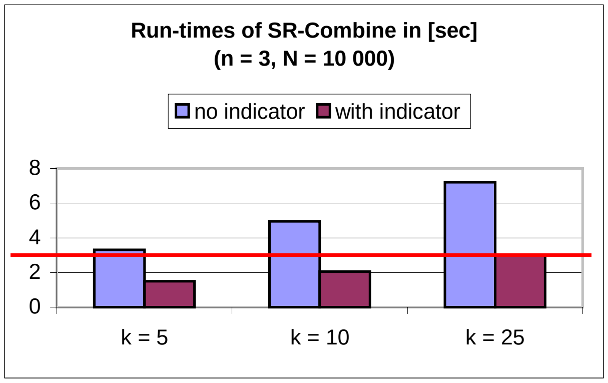

Fig. 2: Effect of indicators on run-times

However, our indicators can perform another important task. Since experiments show that algorithms relying on extensive Random Accesses (like e.g. Quick-Combine) tend to deteriorate in environments where uniformly distributed data prevails, in SR-Combine indicators are used for compensation. If not at least one stream is detected showing a skewed data distribution, no random accesses are granted in-spite of the cost-ratio for the performed sorted accesses. An indicator is said to show a skew, if it is larger than the decrease that can be expected in the uniformly distributed case.

For instance in the case of an equally weighted arithmetical mean as combining function, an indicator that uses the distance between the last and the p-th last object and N databases objects uniformly distributed over the score interval [0, 1], the expected decrease in each stream is given by (p-1 / N). Any larger value for an indicator hints at a skew in the very stream. This simple heuristic takes care that our SR-Combine does not deteriorate, even in cases with uniformly distributed data streams only. ∎

Heuristic 3: Accessing the Most Promising Objects
Having seen some objects by sorted access the algorithm can spend some random accesses granted according to the cost-ratio. However, in most cases there are not enough random accesses to analyze every object and calculate its aggregated score. Thus we have to spend our random accesses wisely for the most promising objects only.

This is implemented by performing random accesses on one of those objects having the maximum lower bound estimation *max_low* first. The completion of the score for these objects will help to increase *max_low* most quickly, which of course propels an early termination of the pruning phase. And the less objects we access during the pruning phase, the less barriers we have to cope with.

If all the objects having the maximum lower bound are completely known, we will use our random accesses on those objects having the maximum upper bound estimation. Since we don't want to keep a list of the current upper bounds (that may change for most objects after each sorted access), we will use a simple heuristic and access objects in the order, they first have been seen. Because obviously those objects seen quite early have the best chances to get a high upper bound estimation. ∎

Having stated our helpful heuristics, we still have to show, that our pruning is correct. Therefore we state the following theorem:

Theorem 1: Correct Termination of the Pruning Phase
If there is at least one object o_x whose lower bound estimation (*max_low*) is larger or equals the threshold calculated using the minimum score values of each stream as input for the monotonic combining function F, no object that has not been seen, can be better than all seen objects, i.e. an overall best object has already been seen.

Proof:
It is obvious that if an object's lower bound for its aggregated score is greater or equal any other object's upper bound, it has to be an overall best object. Thus, what has to be shown is that an overall best object has already been seen as the algorithm terminates the pruning phase. We know that at least one object o_x has been seen, whose lower bound is larger or equals the threshold. Due to the sorting of the streams we also know that each score of an unseen object o is smaller or equal the minimum of score values $s_i(o_i)$ seen in each stream so far. Thus the aggregated score of o is limited by the score of o_x:

$$F\,(s_1(o),...,s_n(o)) \leq F\,(\min\,(s_1(\mathrm{o}_1)),..,\min(s_n(\mathrm{o}_n))) =: \mathit{thres}$$
$$\leq \mathit{low}\,(o_x) \leq F\,(s_1(o_x),\ldots,s_n(o_x))$$ ∎

Now we are ready to present the complete algorithm for our pruning phase. Please note, that no expensive updates of all seen objects' upper bounds are necessary during our phase 1.

The Pruning Phase (Phase1)

```
While ($thres > max\_low$) do
1.   Get a new pair ($o_{new}$, $s_i(o_{new})$) by sorted access on
     stream $i$ and calculate a new indicator $\Delta_i$
2.   If there is at least one indicator showing a skew, set
     $random := random + costratio$.
3.   Update $\min(s_i(o))$ with $s_i(o_{new})$) and calculate the
     threshold $thres$ using the minimum score values of
     each stream as input for $F$.
4.   If $o_{new}$ has already occurred in the index then
     4.1. Update its score entry $s_i(o_{new})$) and recalculate
          its lower bound $low(o_{new})$.
5.   else
     5.1. Initialize a record for $o_{new}$ in the index, initialize
          its score $s_i(o_{new})$) and calculate its lower bound
          $low(o_{new})$.
6.   If $max\_low < low(o_{new})$ set $max\_low := low(o_{new})$.
7.   While $random \geq 1$ and ($thres > max\_low$) do
     7.1. If the score of any object $o$ having
          $low(o) = max\_low$ is not already entirely known,
          perform a random access on object $o$, set
          $random := random - 1$ and (if necessary) update
          $low(o)$ and $max\_low$.
8.   While $random \geq 1$ and ($thres > max\_low$) do
     8.1. Perform a random access on the object $o$ that
          was the earliest object seen by sorted access and
          whose score is not already entirely known, set
          $random := random - 1$ and (if necessary) update
          $low(o)$ and $max\_low$.
9.   Set $i := m$, with $\Delta_m = \max\,\{\Delta_j \mid 1 \leq j \leq n\}$
```

3.3 Identifying the barriers

After we can guarantee the existence of at least one result object due to the pruning, we will identify all barriers out of the seen objects, that still prevent the immediate output. To get a more user-friendly output behavior we will rely on the following heuristic.

Heuristic 4: Using the (Top k)* Retrieval Model
In the following, we will implement the (Top k)* strategy from 2.1 by choosing a subset of barriers, called the working barriers. Working barriers will be chosen as those object having the highest upper bound estimations. Only upper and lower bounds of the barriers in the working barriers set will be updated regularly, i.e. (*k-returned*) objects. Whenever during phase 3 an object is removed from working barriers, one of the remaining barriers is chosen as replacement. ∎

Another theorem will help us, to divide all objects seen into barriers and non-barrier objects.

Theorem 2: Correctness of Identified Barriers
Only objects having an higher upper bound than *max_low* can have a better aggregated score, than the objects with maximum lower bound *max_low*, i.e. only those objects are barriers.

Proof: Obvious due to construction of the estimations. ■

The Identifying Barriers Phase (Phase 2)

1. Iterate the index of seen objects and update for each object *o* the upper bound for its aggregated score *upp*(*o*) using its known score values or –if $s_i(o)$ is unknown for any *i*– the minimum score seen by sorted access in stream *i* as input for *F*.
2. Consider the object o_{top} having the maximum lower bound *max_low*. If more such objects exist, choose one having a highest upper bound.
3. Initialize a list *barriers*, in which all the objects having an upper bound higher than *max_low* (excluding o_{top}) are contained with *oid*.
4. Choose a list *working_barriers* containing those (*k* – *returned*) objects from barriers that have highest upper bounds. Remove the chosen objects from *barriers*.

3.4 Removing the Barriers and Output of Objects

For the last phase we will again use our heuristics from phase 1 and 2. If we remove enough barriers we can output an object guaranteed to belong to the top *k* objects:

Theorem 3: Correctness of Output Objects
Let *returned* be the number of objects already returned. If there are less than (*k* – *returned*) barriers left in *working_barriers* (i.e. phase 3 terminates), the object o_{top} having the maximum lower bound *max_low* is one of the *k* overall best objects.

Proof:
If any object is removed from *working_barriers*, in step 3 of phase 3 the list is filled periodically by using suitable objects from the list *barriers*. If phase 3 has terminated, there must be less than (*k* – *returned*) objects in *working_barriers* and no more objects in *barriers*. Thus only those objects in *working_barriers* and all objects that have already been output can possibly have a better score than o_{top}. Hence there are at most ((*k* – *returned* –*1*) + *returned*) = (*k*–*1*) objects that can have a better score and we can safely output o_{top} as one of *k* best objects.

It remains to be shown, that phase 3 terminates at all. But since *max_low* monotonically increases and the objects' upper bounds in *barriers* and *working_barriers* monotonically decrease, the number of objects in *working_barriers* and *barriers* is steadily decreasing during phase 3. At the latest all lists would definitely be cleared, if all seen objects would have been completely determined by sorted or random access. ■

During phase 3 we will successively remove barriers by sorted or random access. We will again chose most promising candidates for random access. If sometimes no random accesses are available, we use sorted accesses and grant some more random accesses according to our costratio. If the lower bound of any object gets larger than *max_low*, this object becomes our new top object and *max_low* is updated. For all updates we will only focus on the *working_barriers* subset and remove all objects whose upper bound sinks below the maximum lower bound *max_low*. Whenever objects are removed from *working_barriers*, new barriers are inserted from *barriers*, until *barriers* is empty.

The Removing Barriers Phase (Phase 3)
While the number of objects in *working_barriers* is (*k* – *returned*) do

1. If (*random* ≥ 1) then
   - 1.1. Perform a random access with respect to any missing stream on any object *o* in the list *working_barriers* having the highest upper bound. Set *random* := *random* –1.
   - 1.2. Recalculate the upper and lower bound for object *o* in the index.
   - 1.3. If the new upper bound of *o* becomes smaller than or equals *max_low*, remove *o* from *working_barriers*.
   - 1.4. If the new lower bound of *o* becomes larger than *max_low*, update *max_low* and o_{top}. Remove all objects from *working_barriers* whose upper bound is smaller or equals *max_low*.
2. else
   - 2.1. Get a new pair (o_{new}, $s_i(o_{new})$) by sorted access on stream *i*. Update o_{new} and its lower bound *low*(o_{new}) in the index as shown in phase 1. Recalculate the indicator for stream *i*.
   - 2.2. If there is at least one indicator showing a skew, set *random* := *random* + *costratio*.
   - 2.3. If *low*(o_{new}) > *max_low* then
     - 2.3.1. If *upp*(o_{top}) > *max_low*, insert o_{top} in *barriers*.
     - 2.3.2. If o_{new} is in *working_barriers* then remove o_{new} from *working_barriers*, else remove o_{new} from *barriers*.
     - 2.3.3. Update *max_low* and o_{top} with *low*(o_{new}) and o_{new}.
     - 2.3.4. Remove all those objects from *working_barriers*, whose upper bound is smaller or equals *max_low*.
   - 2.4. Recalculate the upper bounds *upp*(*o*) of all objects *o* in *working_barriers* (like in phase 2). If *upp*(*o*) ≤ *max_low*, remove *o* from *working_barriers*.
3. While the number of objects in *working_barriers* is less than (*k* – *returned*) and there are still objects in *barriers* do

3.1. Choose any object from *barriers* and update its upper bound (like in phase 2 step 4).

3.2. If the object's upper bound is larger than *max_low* move it from *barriers* to *working_barriers,* else remove it from *barriers*.

3.5 The *SR-Combine* Algorithm

Now we are ready to present the complete *SR-Combine* algorithm. This algorithm adapts itself closely to any CSA environment and chooses wisely how many sorted and random accesses should be performed to optimize the runtime characteristics. It is also interesting to note that our algorithm contains previous approaches as special cases:

Proposition 1: (proof omitted)
With N as the number of database objects and n as the number of streams it can be stated:

a) By choosing the cost-ratio as (n-1) *SR-Combine* simulates the Quick-Combine approach (without heuristic 4).
b) By choosing the cost-ratio less than $1/(n \cdot N)$ the *SR-Combine* algorithm behaves like Stream-Combine.

To prepare *SR-Combine* we first need an initialization where all data structures and variables are initialized. We take the input parameters n, k and the monotonic combining function F from the user's query (SR-Combine is designed for any monotonic combining function) and get a suitable cost-ratio from the service provider (cf. section 4.1). Information about all objects seen by sorted access has to be maintained: thus create an adequate index structure (e.g. hash-table) to manage oids and score values ordered by oids. For each new oid the structure will contain a record of an array of length n for its single score values, its aggregated score's upper bound and the respective lower bound. Initialize two variables *thres* $\in$ [0,1] with 1 and *max_low* $\in$ [0,1] with 0. Initialize an integer i := 1, counters for the number of possible random accesses *random* := 0 and the number of objects already returned *returned* := 0. Create an array of length n to hold the current minimum scores for each stream and initialize $\min(s_i(o)) := 1$ $(1 \le i \le n)$.

```
Algorithm SR-Combine (F, n, k, cost-ratio)
While less than k objects have already been returned, i.e. returned < k, do
1. If the index is not empty, iterate the index and get the best lower bound, i.e. max_low := max(low(oid)).
2. Compute the threshold thres by using the minimum score values of each stream as input for F.
3. Perform the "Pruning Phase".
4. Perform the "Identifying Barriers Phase".
5. Perform the "Removing Barriers Phase".
6. Output $o_{top}$ as one of the k best objects, increase returned by one, mark $o_{top}$ in the index as finished (it must not be accessed again).
```

4. Performance Benchmarks

4.1 Determining the Cost-ratio

An important factor to run *SR-Combine* is the cost-ratio introduced by [FLN01]. It determines the ratio between average costs for sorted and random accesses for the specific application and environment. Of course these costs depend on various influences, like the speed and average workload of the central server, network latencies, external data sources involved and last, but not least the specific query. Most of these influences cannot be estimated by the user, but have to be determined by the service provider. Since services are designed for a specific application, it's generally quite easy to anticipate user interaction. Statistical analysis thus helps to get a set of typical cost-ratios that can be assigned to different queries.

Generally speaking, different streams may be provided by different sources or subsystems. Some of them might not allow random accesses, others probably forbid sorted accesses and even those, who admit both types may do this at different costs (in terms of money and/or time). Hence it may be worthwhile to consider costs separately for each stream and calculate n specific cost-ratios. In this way we can optimize choice between both kinds of accesses in every stream much more precisely. Though it is not really complicated to refine the algorithm along these lines (n different counters for sorted and random access have to be initialized and updated according to the indicator's choice of stream), we will assume an average cost-ratio throughout this paper to simplify matters.

In the following for the average cost-ratio we will use the experimentally determined factor of 6 SA=1 RA from our scenario 3 in all experiments for comparability reasons. However, we also tried different variations of the cost-ratio leading to very similar results (cf. section 4.4).

4.2 Test Environment

For our tests we will focus on both *indirect measures* given by the number of necessary object accesses and *direct measures* concerning the real-time capabilities of our algorithm in practical applications. But first we have to look for some suitable algorithms as a benchmark for *SR-Combine*. Of course one candidate is always the naïve approach that shows its advantages especially in small databases due to low CPU costs and thus nearly performs in constant time. Performance tests in our experimental environment show that algorithms of the Quick-Combine or CA type optimize the necessary object accesses and perform best in environments with skewed data. However, they tend to deteriorate in efficiency, if large amounts of uniformly distributed data is involved (for n=5, k=25 CA and QC are already four times worse than the naïve approach!). Another algorithm is given by the Stream-Combine or NRA type which behave in the opposite way and are useful only for uniformly distributed data.

We will show this behavior as an example for the case of three streams to combine and for different numbers of objects to return (see diagrams in fig. 3 and 4), but the results even get worse for higher numbers of streams. Though all of these algorithms gain their highest performance only in special cases, we will accept the challenge and compare *SR-Combine* to Quick-Combine and CA for skewed data and to the naïve approach and Stream-Combine in uniformly distributed environments.

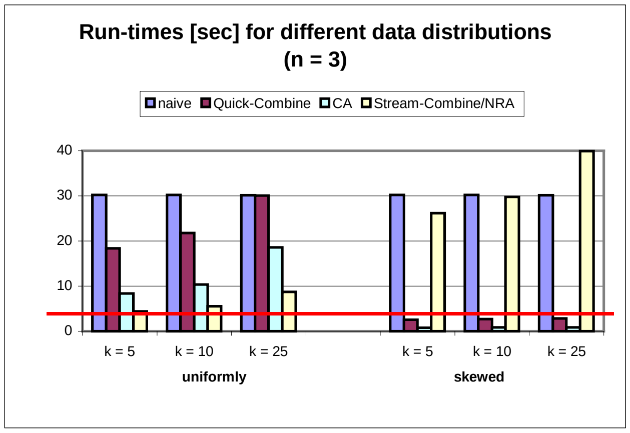

Fig. 3: Deteriorating efficiency in different environments

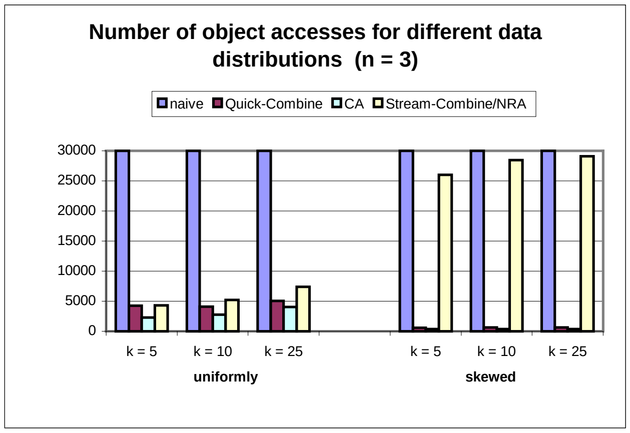

Fig. 4: Object accesses for run-times in fig. 3

In the diagrams we focus on run-times (in seconds) and object accesses (sorted and random accesses cumulated). On the left-hand side of the diagrams, the results for uniform distributions are given, on the right-hand side results for skewed data. Please note that though in the uniformly distributed case CA and Quick-Combine use even less accesses than Stream-Combine, the run-times are worse, due to their extensive use of expensive random accesses.

As stated in section 2 the target architecture for *SR-Combine* is of the CSA kind. Thus our following tests have been performed using a Java middleware environment (Sun JDK 1.3) running on a 600 MHz Pentium II PC with 256 MB RAM as application server connected via a 100 MBit LAN to an IBM DB2 V7.2 Universal Database Server for random and sorted accesses. We used JDBC for all accesses, sorted accesses are performed via Java result sets, random accesses simulated with use of temporary tables (which allowed to change to cost-ratio for tests in section 4.4). The statistically independent data sets for our experiments contain $N = 10000$ database objects; their scores are generated synthetically according to different practical data distributions [GBK00]. Also using the IBM DB2 Database the cost-ratio between random and sorted accesses was determined statistically to 6 SA=1 RA with 1 SA≈1 msec (i.e. 1 RA≈ 6 msec).

4.3 *SR-Combine* vs. others

Having set up the benchmark environment we performed several tests. The diagrams below show statistical averages of the experimentally determined run-times and total numbers of object accesses (SA+RA). Basically we have set up two scenarios one focussing on uniformly distributed data, the other one involving skewed data. In both scenarios we tested all algorithms in the case of three streams combined (left-hand side) and five streams combined (right-hand side) for different practical values of *k* (*k* = 5, 10, 25). Again the horizontal line shows our target of up to 3 seconds run-time [Pop97].

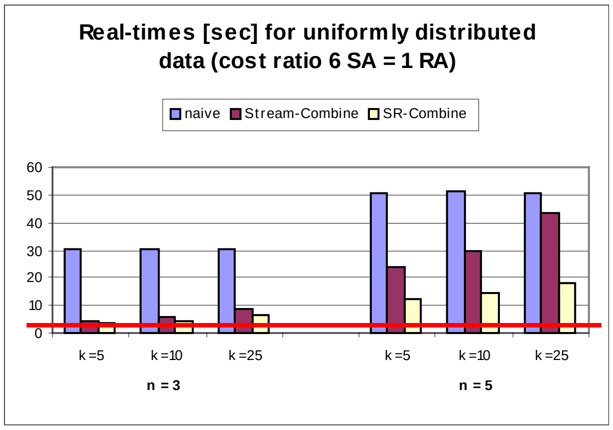

Fig. 5: Benchmark for uniform data distributions

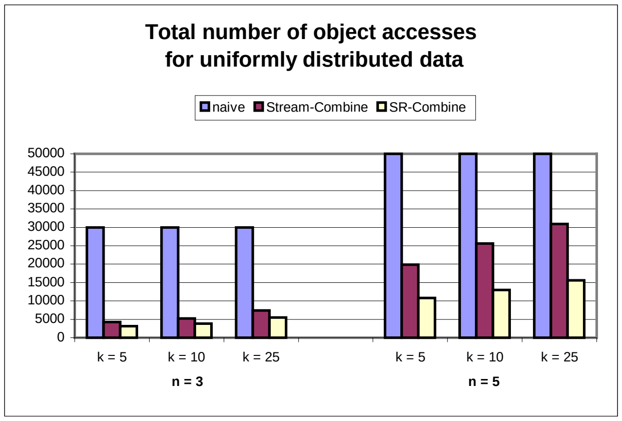

Fig. 6: Object accesses for the benchmark in fig. 5

As can easily be seen (fig. 5 and 6), in the case of uniformly distributed data the *SR-Combine* algorithm performs always much better than the naïve approach and even always better than Stream-Combine the best algorithm known in this environment. For the tests where skewed data was involved, we left out the naïve approach, because today's algorithms perform much better and the scale would be affected (the respective runtimes for the naïve approach are 30 and 50 seconds). As we can see (fig. 7 and 8), also in the case involving skewed data, *SR-Combine* beats both state of the art algorithms. Please note that the *SR-Combine* algorithm has been run in all of these tests (uniform and skewed data) always with the *same* set of input parameters (*F*, *n*, *k*, cost-ratio). It has automatically adapted itself to the respective situation.

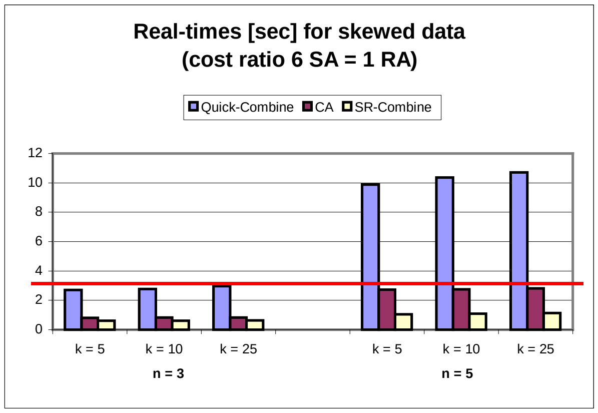

Fig. 7: Benchmark for skewed data

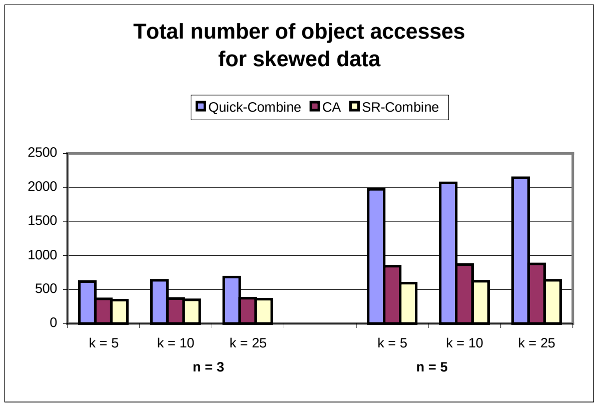

Fig. 8: Object accesses for the benchmark in fig. 7

SR-Combine in both scenarios obviously not only essentially improves the run-times, but also optimizes the total object accesses by wisely choosing the suitable kind of access. The improvement factor grows with the number of streams to combine and increasing numbers of objects to return. It also scales with the size of the database. In the case of skewed distributions our indicator-based heuristic even improves Quick-Combine and CA, that are shown to minimize the number of objects accessed [FLN01].

For skewed data *SR-Combine* already shows real-time capabilities. This is important, because [GBK00] give evidence that this is the case in most real-world applications. However, in the case of uniform data distributions, *SR-Combine* still fails to meet real-time requirements.

4.4 Adaptability and Scalability

We already explained the use of cost-ratios. However, what influence have different cost-rations on the *SR-Combine* algorithm? The basic idea is that our algorithm replaces random accesses with some more sorted accesses in environments where random accesses are expensive. In the following diagram we see run-times for different database sizes with different cost-ratios applied (using delays to slow or speed up accesses). We have taken all the different cost-ratios from our three application scenarios. Obviously SR-Combine adapts well to different environments, thus the total run-time changes only slightly for different cost-ratios (cf. fig. 9). Besides the algorithm scales well with growing database sizes. (For space reasons we only present the skewed case. However, the uniform case shows a very similar behavior except that the deviation of times is a little higher).

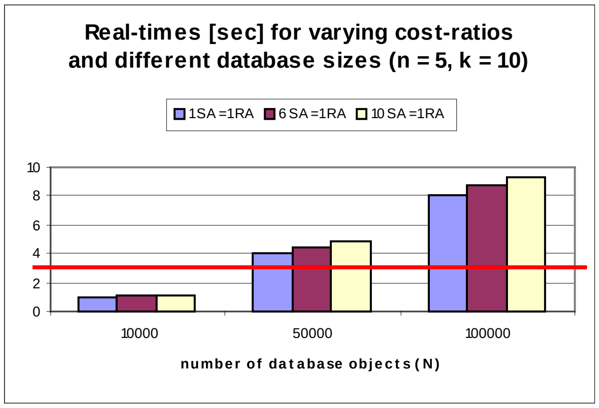

Fig. 9: Variations of cost-ratios and database sizes

4.5 Mobile Real-time Considerations

In contrast to e.g. the naïve approach the (top k)* retrieval model allows the earliest possible output of objects, while the algorithm is still running. Especially in mobile environments this is a useful feature, because the available bandwidth can be used more efficiently. Besides, we need response times up to 3 seconds to satisfy service users. We have seen that in our environments involving skewed data, *SR-Combine* has no problem to meet these requirements, but if scores show only a small skew or even uniform distributions this may not be the case. However, if an object is delivered every couple of seconds, psychologically users will not notice the waiting period until all requested objects have been returned. Thus though the total running time may exceed 3 seconds the mobile application requirements can psychologically still be met.

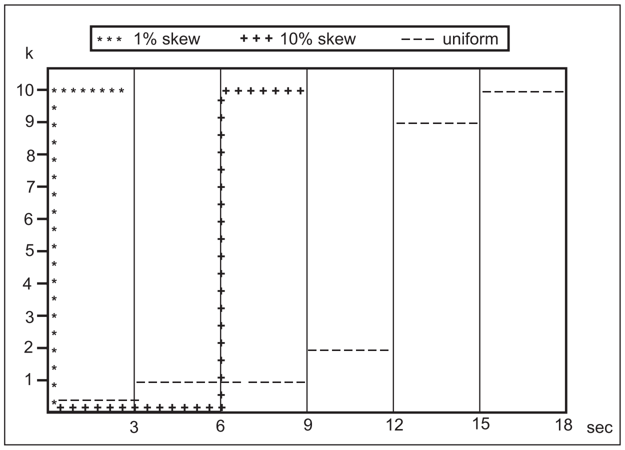

Fig. 10: Output behavior for different distributions

The diagram in fig. 10 shows the return behavior of our algorithm for different skews combining five streams (*n*=5). If a total of ten objects has to be returned (*k*=10), the graphs show how many objects have already been delivered during each 3 second time-span. Obviously the total run-times increase towards the uniform case. Although only the very skewed distribution meets the hard real-time requirements, please note that even in case of uniform distributions (with a total run-time of 18 seconds) *SR-Combine* delivers some first results early thus improving the subjective waiting time for users [Pop97]. This leads to an at least "acceptable" case, for those environments where run-time constraints cannot be met yet.

4.6 Lessons Learned for Mobile Applications

- Algorithms for (mobile) top *k* querying have to focus on both: access costs and CPU costs. Choosing the right stream for access will also improve the average run-times.
- Since run-times can be improved by wisely balancing sorted and random accesses, a competitive algorithm has to self-adapt to different environments characterized by different access costs for different services.
- Psychologically the expected response time is up to 3 seconds, but if that is not possible, delivering already parts of the result every 3 seconds helps to improve subjective waiting times.

Our benchmark results show that *SR-Combine* already delivers results within the acceptable time-span of 3 seconds for database sizes of 10000 to 50000 objects, if skewed data is involved and the average cost for a sorted access is at most 1 msec. We also pointed out that the retrieval model for successive output at least psychologically helps in other cases to satisfy users. But what can be done to further improve the real-time performance in cases where the requirements for mobile services are not yet met?

As we stated, a sorted access in our current prototype system costs about 1 msec. But the access costs and the cost-ratios between sorted and random access strongly depend on the kind of service and the system environment. Thus we have two candidates for improvements. Either the system's hardware can be improved guaranteeing faster object accesses, or software techniques can be applied to essentially speed up sorted and random accesses in the database e.g. by using suitable multidimensional indexes like in [BM+01, BB+01].

We have shown that *SR-Combine* essentially reduces the object accesses and is able to cope with a wide range of service-dependent cost-ratios. Thus, if a specific service is projected and its cost parameters are known, a custom-made hardware/software selection can be designed to fit the requirements of both service provider (low hardware costs, high flexibility/extensibility, etc.) and users (low service costs, low waiting times, etc.). Consider for example a service provider needing an average environment like in our uniformly distributed case (e.g. *n*=5, *k*=10, *N*=10000, cost-ratio 6 SA =1 RA). If a sorted access could be brought down to 0.2 msec instead of 1 msec, the service using *SR-Combine* would already meet real-time requirements.

5. Summary and Outlook

In this paper we addressed the problem of top *k* query processing for applications in mobile services. Having stated the necessary real-time requirements, different application scenarios with typical environment variables and a reference architecture for mobile services, we presented a new self-adapting algorithm *SR-Combine* for top *k* retrieval. We presented benchmarks with current combining algorithms and investigated scalability and adaptability for our algorithm. SR-Combine outperforms the naïve approach and all current algorithms in both object accesses and real-time capabilities.

We focused on real-time capabilities, that are crucial for the acceptance of mobile services. Though we did not meet the hard psychological real-time constraint of at most 3 seconds response time, the algorithm has proven to be robust against changes of access costs and using an advanced retrieval model delivers objects successively. This is psychologically important especially in mobile environments, because users get at least some first objects within the expected response time of 3 seconds. Thus the subjective waiting time is reduced.

Our future work will transfer efficient top *k* querying with *SR-Combine* into practical environments like in our prototypical implementation of [WBK01]. Besides, we will focus on getting more efficient heuristics to balance the use of expensive random accesses between different phases. Though in phase 3 more information is available and expensive accesses could be used more effectively, postponing these accesses from phase 1 to phase 3 may delay an early termination.

6. Literature

[BB+01]
S. Berchtold, C. Böhm, D. Keim, F. Krebs, H-P. Kriegel: On Optimizing Nearest Neighbor Queries in High-Dimensional Data Spaces. In *Proc. of the Intern. Conf. on Database Theory (ICDT'01)*, LNCS 1973, Springer, 2001.

[BGM02]
N. Bruno, L. Gravano, A. Marian: Evaluating Top-k Queries over Web-Accessible Databases. *In Proc. of the Intern. Conf. on Data Engineering (ICDE'02)*, San Jose, CA, USA, 2002.

[BM+01]
K. Böhm, M. Mlivoncic, H-J. Schek, R. Weber: Fast Evaluation Techniques for Complex Similarity Queries. In *Proc. of the Intern. Conf. on Very Large Databases (VLDB'01)*, pp. 211-220, Rome, Italy, 2001

[CG99]
S. Chaudhuri and L. Gravano: Evaluating top-*k* selection queries. In *Proc. of the Intern. Conf. on Very Large* Databases *(VLDB'99)*, Edinburgh, Great Britain, 1999.

[CK97]
M. Carey and D. Kossmann: On saying "Enough Already!" in SQL. In *Proc. of the ACM Intern. Conf. on Management of Data (SIGMOD'97)*, Tucson, AZ, USA, 1997.

[Coh98]
W. Cohen: Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. In *Proc. of the ACM Intern. Conf. on Management of Data (SIGMOD'98)*, Seattle, USA, 1998.

[DGS00]
J. Ding, L. Gravano, N. Shivakumar: Computing Geographical Scopes of Web Resources. In *Proc. of the Intern. Conf. on Very Large Databases (VLDB'00)*, Cairo, Egypt, 2000.

[DR99]
D. Donjerkovic and R. Ramakrishnan: Probabilistic optimization of top queries. In *Proc. of the Twenty-fifth International Conference on Very Large Databases (VLDB'99)*, Edinburgh, Great Britain, 1999.

[Fag96]
R. Fagin. Combining fuzzy information from multiple systems. In *Proc. of the ACM Symposium on Principles of Database Systems (PODS'96)*, Montreal, Canada, 1996.

[FLN01]
R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the ACM Symposium on Principles of Database Systems(PODS'01)*, Santa Barbara, CA, USA, 2001.

[GB97]
S. Gribble and E. Brewer. System design issues for internet middleware services: deductions from a large client trace. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, 1997

[GBK00]
U. Güntzer, W-T. Balke, and W. Kießling: Optimizing multi-feature queries for image databases. In *Proc. of the Intern. Conf. on Very Large Databases (VLDB'00)*, Cairo, Egypt, 2000.

[GBK01]
U. Güntzer, W-T. Balke, and W. Kießling: Towards efficient multi-feature queries in heterogeneous environments. In *Proc. of the IEEE Inter. Conf. on Information Technology: Coding and Computing (ITCC'01)*, Las Vegas, USA, 2001.

[GW00]
R. Goldman and J. Widom: WSQ/DSQ: A practical approach for combined querying of databases and the web. In *Proc. of the 2000 ACM Intern. Conf. on Management of Data (SIGMOD'00)*, Dallas, TX, USA, 2000.

[NR99]
S. Nepal and M. Ramakrishna: Query processing issues in image (multimedia) databases. In *Proc. of the Intern. Conf. on Data Engineering (ICDE'99)*, Sydney, Australia, 1999.

[OR+98]
M. Ortega, Y. Rui, K. Chakrabarti, K. Porkaew, S. Mehrotra, and T. S. Huang: Supporting ranked boolean similarity queries in MARS. *In IEEE Transactions on Knowledge and Data Engineering (TKDE)*, Vol. 10 (6), 1998.

[Pop97]
E. Pöppel, A hierachical model of temporal perception. In *Journal of Trends in Cognitive Science*, Vol. 1, Elsevier, 1997

[TSH01]
M. Tork Roth, P. Schwarz, L. Haas: An Architecture for Transparent Access to Diverse Data Sources. In *Compontent Database Systems*, 2001.

[WBK01]
M. Wagner, W-T. Balke, and W. Kießling: An XML-based Multimedia Middleware for Mobile Online Auctions. In *Proc. of the Intern. Conf. on Enterprise Information Systems (ICEIS'01)*, Setubal, Portugal, 2001.

[WH+99]
E. Wimmers, L. Haas, M. Tork Roth, C. Braendli: Using Fagin's Algorithm for Merging Ranked Results in Multimedia Middleware. In Proc. of the Intern. Conf. on Cooperative Information Systems (CoopIS'99), Edinburgh, Great Britain, 1999.