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. We present a new algorithm SR-Combine that closely self-adapts to particular cost ratios in different environments optimizing both object accesses and query run-times. We perform a series of benchmarks to verify the superiority over existing approaches and use a psychologically founded model of response time requirements for mobile access. For a wide range of practical cases SR-Combine can already satisfy these goals. Where this isn't yet the case, we show ways to get there systematically paving the way for real-time capabilities in mobile services.

1. Introduction

Top *k* querying is an increasingly demanding problem in today's applications. Areas like web-based information services, enterprise information systems or content-based retrieval already make intensive use of this new paradigm. A variety of applications for ranking preferences in databases [Kie02], multi-media databases [OR+98] or recent approaches in cooperative 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 needs to know about recent similar cases or some 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."

Here 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 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 online search engines and users will allow higher run-times only for very difficult tasks (e.g. in work environments). In mobile applications there are even monetary reasons involved, because the connection is often charged with respect to the usage time.

Recent approaches to top k querying were designed for special applications and tended to deteriorate in efficiency very quickly, if the application environment or score distributions for 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– run-times will nevertheless considerably increase, if the algorithms extensively use expensive types of object accesses, cf. [WH+99]. Thus also the different costs for types of accesses and the application architecture have to be considered to build competitive algorithms. Due to the technical constraints of mobile devices in the following we focus on few streams to combine ($n < 10$), small numbers of return values ($k = 3$ to 10), and on response times of only few seconds.

2. Top k Querying for Mobile Services

2.1 Characteristics of Mobile Services

With the current developments of devices like cell phones or PDAs, pervasive access on information becomes more and more attractive. We will illustrate this by three scenarios of mobile services and discuss specific characteristics and requirements.

- **Scenario 1: Mobile access to global stock exchanges**. Typical queries are: ”Get the five top performers in IT industries also involved in bio-technology.” The querying is mainly attribute-based and there is virtually no difference in costs of sorted accesses and random accesses (cf. [NR99]).
- **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 often uses nearest neighbor searches. 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 having the lowest prices.” The querying involves attribute-based parts often together with extracted multimedia features. Random accesses are more expensive than sorted accesses (tests in [WBK01] show factors of about 6).

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 data sources. Whereas attribute-based stock market information has update ranges of few seconds, the information for complex services involving e.g. multimedia data, is more durable. Typi-

cal update intervals range between days and weeks. This means that e.g. stock market information has to be imported directly from content providers. But durable information like multimedia data can be transferred on central servers allowing efficient storage and indexing schemes. The research area of data integration over the web [GB97, TSH01, GW00] has lead to twofold architectures for different applications:

- **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 and content provider differ or short update ranges are necessary, services are provided using a middleware gathering information on demand from sources accessed via the Internet.

Enabling top *k* queries in mobile environments, however, poses severe problems. In [BGM02] ways to build mobile services with a DSA architecture providing direct access to various Internet sources are presented, however their tests show that the processing even for simple tasks often need hours. Thus when trying to meet real-time requirements today, we have to build all scenarios on a CSA architecture. A solution of combining Internet sources with local database servers is given by the WSQ/DSQ approach [GW00] that handles accesses to Internet sources in an asynchronous manner and caches the results for later use in virtual tables of a central database. Since the service provider can anticipate the type of queries, commonly accessed data and the update rate needed, a caching strategy with asynchronous updates is suitable.

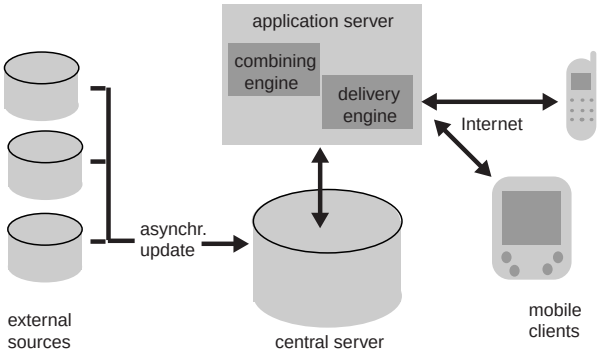

Fig. 1. Intended CSA architecture

The mobile service in Fig. 1 consists of a application server containing a combining engine which runs the *SR-Combine* algorithm. A detailed architectural study is given in [BKU02]. All the content is retrieved from a central server (updated asynchronously). As shown in [WBK01] with the example of mobile online auctions the delivery engine can automatically transform generic XML formats using XSLT to support mobile devices e.g. via WAP or i-mode gateways. Another advantage of this architecture is that data on the central server can be indexed to suit the service design. Through statistical analysis also costs for certain usage patterns can be estimated for different kinds of access personalized for each user.

2.2 Top *k* Querying Revisited

Since top *k* querying is an important feature for cooperative information systems, we will revisit some approaches that can later be used as a yard stick. Due to the nature of mobile services showing high update rates or even distributed sources, we are mainly concerned with middleware algorithms. In general those algorithms can be divided into two categories: the ones guaranteeing a correct result set and those using statistical data to get the result set more efficiently, e.g. [DR99], however with an amount of uncertainty about the correctness. Since mobile services with restricted client devices (bandwidth, display-size, etc.) demand rather small numbers of objects to be output, we will insist on a correct result set, i.e. we focus on high precision. Applications for top *k* querying are concerned with gathering information using a variety of classifiers. Generally speaking there are two 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.

Due to the CSA architecture we can rely on both kinds of accesses. Having gathered all basic score values, the total score is subsequently determined using a suitable monotonic combining function *F*. Both access methods can however essentially differ in their costs. The rule of thumb can be stated that the more complex the information to be queried, the more expensive random accesses will be. This leads to various applications 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 return a correct result set. A first algorithm was given by [Fag96] followed by improved approaches for multimedia retrieval [OR+98] and applications in fuzzy logic [NR99]. [GBK00] generalized a threshold algorithm and improved it by heuristic control flows in the Quick-Combine approach. Though it was proven to be optimal in minimizing sorted accesses for top *k* querying [FLN01], extensive random accesses had to be performed. Since costs for random accesses may explode in some applications, it can be far more economic 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 without random accesses. However, in practical tests those algorithms showed their limited applicability by accessing 80-90% of all database objects, if skewed data is involved, thus quickly loosing their speed-up even over the naïve approach. Extensive performance tests in [GBK00] show that Quick-Combine achieves its best results, if skewed data is involved, whereas Stream-Combine performs best in environments with uniformly distributed data. To overcome these limitations an combined algorithm called CA is given in [FLN01]. Basically CA runs NRA, but periodically performs random accesses like Quick-Combine on promising objects. However, it shows the same characteristic behavior and limitations as Quick-Combine. Thus a mere combination of known algorithms will not solve the problem, but we have to design a new paradigm to get a competitive self-adapting algorithm.

3. The *SR-Combine* Algorithm

In the following we will present the *SR-Combine* algorithm returning k overall best objects. We will first present a sketch of its three phases, state an adequate retrieval model and present heuristics leading to an efficient implementation of the algorithm.

Algorithm *SR-Combine*

- **Phase 1 (Pruning Phase):** *Phase 1 gathers objects from the different streams until it can guarantee that at least one overall best object has been seen. It uses sorted accesses to see object scores in descending order in each stream and random accesses to complete the scores for objects already seen. For each object there are lower and upper bound estimations for its total score.*
- **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, the so-called barrier objects. After phase 2 we know that the overall best object is among the barrier objects and all the non-barrier objects can be excluded in our search for the top object.*
- **Phase 3 (Removing Barriers):** *Phase 3 successively completes scores of barrier objects. Again both sorted and random accesses are used. Whenever the information about any object is sufficient to exclude it from the barriers, the object is removed. If enough barrier objects have been removed, the algorithm can output the top object and start over to get the next best object.*

3.1 A Faster (Top k)* Retrieval Model

Since the correctness of the result set is guaranteed, 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 output, if we are positive that it belongs to the set of k best matches, no matter what its exact rank or score in the final result set will be. Thus, our (top k)* retrieval model outputs the k best objects, but does not initially determine whether an object will be the first ranked or the k-th ranked object. Though 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 important for psychological reasons and allows an efficient use of bandwidths. Besides users get an idea of the result set and can decide, if the results are already satisfying and the algorithm can be terminated early or if a query has to be refined.

In our case the implementation of this retrieval model is the assertion that we can already output the first object, if there are only (k-1) barrier objects left, i.e. only (k-1) objects have a chance of being more relevant than our object. Inductively we can conclude, that if two objects should be output, there may at most be (k-2) barriers, for three objects (k-3) barriers, and so on. This leads to the observation, that when removing barriers for the output of a first object, we may focus on a set of only k objects and only need to consider 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, that have to be updated regularly during the removing of barriers.

3.2 The Pruning Phase

For an efficient pruning phase we now state a few heuristics and prove its correctness.

Heuristic 1: Taking the Environment into Account
The costs for sorted and random accesses may strongly differ depending on the environment [NR99, WH+99, Coh98]. Thus strategies like optimizing the total number of accesses, however using an expensive kind of access will fail. Thus the ratio between sorted and random access costs has to be taken into account. Since some sorted accesses are necessary to see new objects, 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 total runtime costs may at most double, though the total number of objects accesses is still optimized. Thus the trade-off between minimizing total object accesses and ruining the runtime by using more cost expensive accesses is bounded. This technique 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 (skewed data), streams with uniformly distributed score values or even streams that provide similar scores for all objects. Of course the evaluation of *quickly decreasing* streams should be preferred, because they discriminate well. A second important factor to estimate a stream's influence is its weighting in the combining function. *Highly weighted* streams should be preferred. [GBK00] proposes an indicator technique that uses the relative decrease of a stream's score distribution together with a partial derivative of the combining function for the stream (if the derivative exists, otherwise - e.g. in the case of max as combining function - this part can be set to $1/n$). For the relative decrease in each stream we use the 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 number. Indicators for each stream are given by:

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

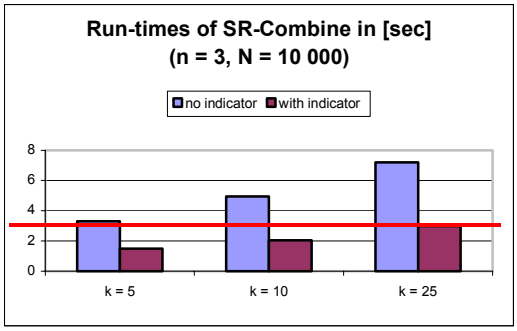

Fig. 2. Effect of indicators on run-times

Always choosing the right stream for sorted accesses leads to an important improvement factor for the algorithm's real time capabilities. Theoretically at most a factor of

n can be reached over classical strategies. Since practical tests (cf. fig. 2) show that even in the case of only three streams involved our indicator gains a factor of 2 (in run-times and object accesses), we adapted this indicator for our algorithm. The line in our real-time diagrams marks the 3 seconds requirement (cf. section 4).
However, SR-Combine's indicators perform another important task. Since experiments show that algorithms relying on extensive random accesses tend to deteriorate in cases 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 expected decrease in the uniformly distributed case. For instance in the case of an arithmetical mean as combining function, an indicator that uses the distance between the last and the *p*-th last object and *N* databases objects, the expected uniformly distributed decrease in each stream is (*p*-1 / *N*). This simple heuristic takes care that SR-Combine does not deteriorate, even in cases with uniformly distributed streams only. ■

Heuristic 3: Accessing the Most Promising Objects
Having seen some objects by sorted access the algorithm can spend random accesses granted according to the cost-ratio. However, there will not be enough random accesses to analyze objects thoroughly. Thus we have to spend our random accesses wisely for the most promising objects only. This is implemented by performing random accesses on those objects having the maximum lower bound estimation *max_low* first. Knowing the score for these objects helps to increase *max_low* quickly, which 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 afterwards. If all objects with 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. ■

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 the 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:

$$F(s_1(o),...,s_n(o)) \leq F(\min(s_1(o_1)),..,\min(s_n(o_n))) =: thres \leq low(o_x) \leq F(s_1(o_x),\ldots,s_n(o_x)) \quad (2)$$

■

Now we 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 Δ_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* ≥ 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* ≥ 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 correct result object, we will identify all barriers. For an improved output behavior we rely on the following heuristic and state a theorem helping to distinguish between barriers and non-barrier objects.

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. Whenever during phase 3 an object is removed from working barriers, one of the remaining barriers is chosen as replacement. ■

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) . ■

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 use again our heuristics from phases 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 among 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/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 no random accesses are available, we use sorted accesses and grant some more random accesses according to our cost-ratio. If the lower bound of any object gets larger than the maximum lower bound *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 *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 $upp(o) \leq max\_low$, 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 closely to any application and chooses wisely how many sorted and random accesses should be performed to optimize the run-time characteristics. It is also interesting that our algorithm contains previous approaches as special cases:

Proposition 1: (proof omitted)
With N as number of database objects and n as number of streams we can state:
1. By choosing the cost-ratio as (n-1) *SR-Combine* simulates the Quick-Combine approach (without heuristic 4).
2. By choosing the cost-ratio less than $1/(n \cdot N)$ the *SR-Combine* algorithm behaves like Stream-Combine. ■

For initialization. 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 the cost-ratio from the service provider (cf. section 4.1). Since information about all objects seen by sorted access has to be maintained, we intialize an adequate index structure (e.g. hash-table) to manage oids and score values ordered by oids. For each new oid the structure contains a record of an array of length n for its single score values, its aggregated score's upper bound and the lower bound. We initialize variables $thres \in [0,1]$ with 1, $max\_low \in [0,1]$ with 0 and integer $i := 1$, counters for the number of possible random accesses $random := 0$ and the number of objects already returned $returned := 0$. We 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. $max\_low := \max(low(\text{oid}))$.
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. It determines the ratio between average costs for sorted and random accesses for the specific application. Of course 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 have to be determined by the service provider. Since services are designed for a specific application, user interaction can generally be anticipated. Statistical analysis thus provides a set of typical cost-ratios.

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 is worthwhile to consider costs separately for each stream and calculate n specific cost-ratios. Though it is not really complicated to refine our algorithm along these lines (n different counters for sorted and random access have to be updated), we will assume an average cost-ratio 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. Of course one candidate is the naïve approach that shows its advantages especially in small databases due to low CPU costs. Performance tests in our experimental environment show that algorithms of the Quick-Combine or CA type 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 for uniformly distributed data only.

We will exemplify this behavior for three streams to combine and 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 algorithms gain their highest performance only in special cases, we will accept the challenge and benchmark *SR-Combine* with Quick-Combine and CA for skewed data and the naïve approach and Stream-Combine for uniformly distributed data. 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.

Our following tests have been performed using a Java middleware running on a 600 MHz PentiumII PC with 256 MB RAM as application server connected via a 100 MBit LAN to an IBM DB2 V7.2 Database Server for random and sorted accesses. We used JDBC for all accesses, sorted accesses are performed via Java result sets, random accesses 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 distributions [GBK00]. 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).

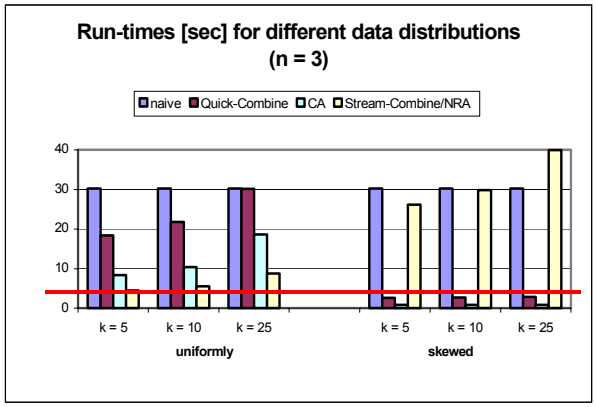

Fig. 3. Deteriorating efficiency in different environments

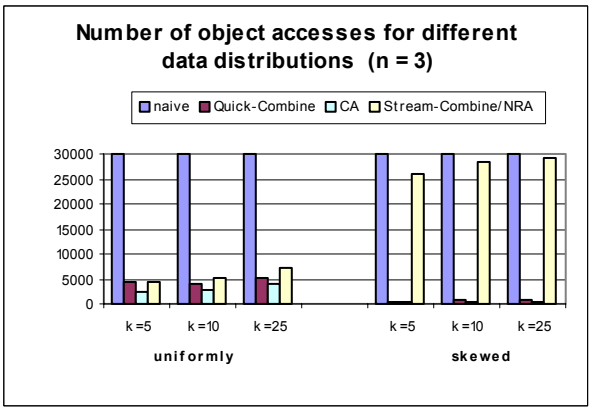

Fig. 4. Object accesses for run-times in fig. 3

4.3 *SR-Combine* vs. others

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

Fig. 5 and 6 show that 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 on skewed data, we left out the naïve approach, because the scale would be af-

fected (respective runtimes for the naïve approach are 30 and 50 seconds). Fig. 7 and 8 show that 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.

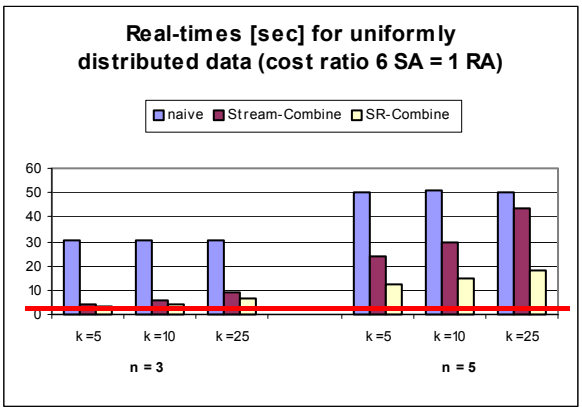

Fig. 5. Benchmark for uniform data distributions

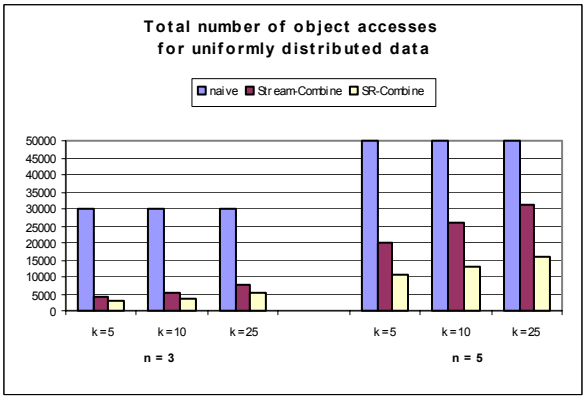

Fig. 6. Object accesses for the benchmark in fig. 5

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 minimize the number of objects accessed.

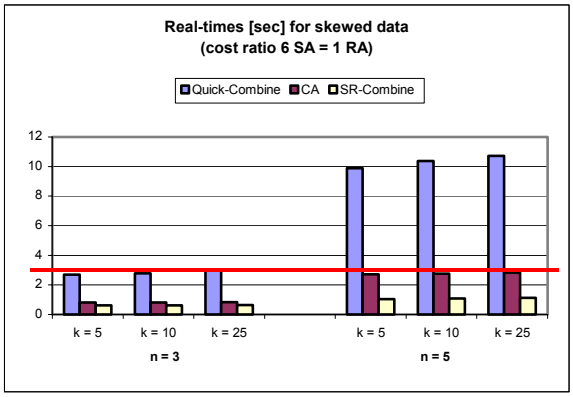

Fig. 7. Benchmark for skewed data

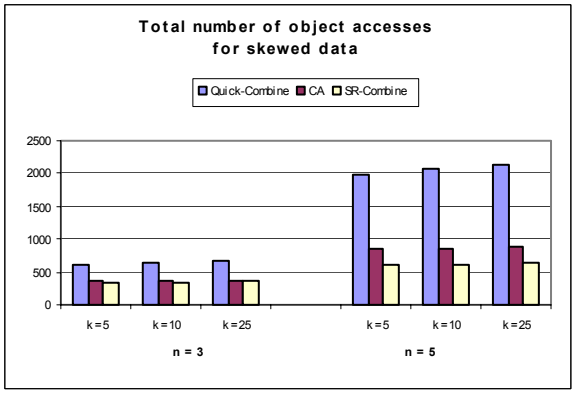

Fig. 8. Object accesses for the benchmark in fig. 7

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 applications where random accesses are expensive. The following diagram shows run-times for various database sizes with different cost-ratios applied (using delays to slow or speed up accesses). We tested all different cost-ratios from our three application scenarios. Obviously SR-Combine adapts well to different applications, 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).

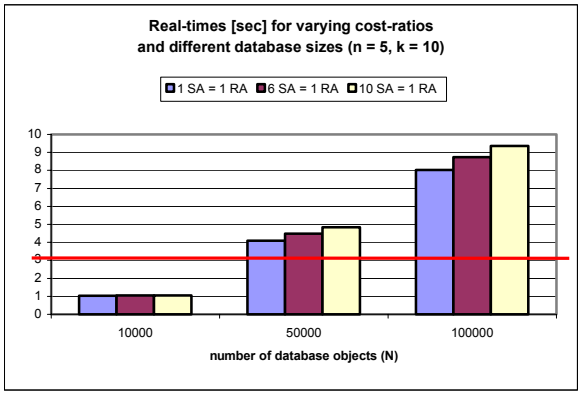

Fig. 9. Variations of cost-ratios and database sizes

4.5 Mobile Real-time Considerations

The (top k)* retrieval model allows the earliest possible output of objects. Especially in mobile environments this is useful, because the available bandwidth can be used efficiently. But needing response times up to 3 seconds, we have seen that in environments involving skewed data, *SR-Combine* can meet these requirements, but fails if scores show uniform distributions. However, if an object is delivered every couple of seconds, users tend to accept the waiting period until all objects have been returned. Thus the real-time requirements can psychologically still be met. The diagram in fig. 10 shows the output behavior for different skews combining five streams (n=5) with ten objects to return (k=10). The graphs show how many objects are output during each 3 second time-span. Though only very skewed distributions meets the hard real-time requirements, please note that even for uniform distributions (with a total run-time of 18 seconds) *SR-Combine* delivers some first results early. Thus the subjective waiting time is improved leading to at least acceptable cases.

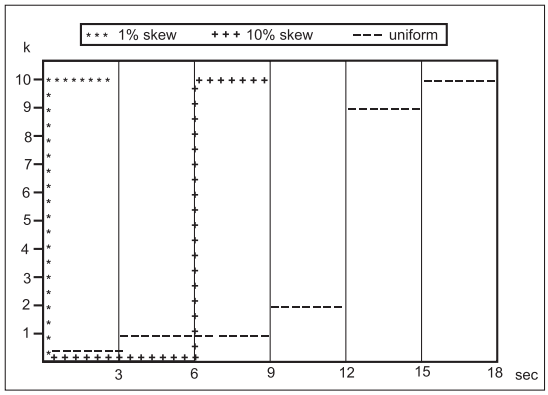

Fig. 10. Output behavior for different distributions

4.6 Lessons Learned for Mobile Applications

- Algorithms for (mobile) top *k* querying have to focus on both access costs and CPU costs. Choosing right streams for access will also improve average run-times.
- Run-times can be improved by balancing sorted and random accesses. A competitive algorithm has to adapt to applications characterized by different access costs.
- Psychologically the acceptable response time is about 3 seconds, but delivering already parts of the result every 3 seconds improves subjective waiting times.

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

The access costs and the cost-ratios between sorted and random access strongly depend on the kind of service and the system environment. Thus either the hardware can be improved, or techniques can be applied to essentially speed up database accesses e.g. using multi-dimensional indexes like [CB+00, BM+01]. We have shown that *SR-Combine* essentially reduces the object accesses and copes 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 service provider (low hardware costs, high flexibility/extensibility, etc.) and users (low service costs, waiting times, etc.).

5. Summary and Outlook

We presented a new self-adapting algorithm *SR-Combine* for top *k* retrieval and focused on its necessary real-time requirements for mobile services. We investigated different scenarios with typical environment variables and a reference central server architecture. Since *SR-Combine* flexibly adapts to various access costs, a migration to distributed service architectures is straightforward, if real-time constraints are supported by higher bandwidths or faster networks (e.g. using UMTS). We presented benchmarks with current middleware combining algorithms and showed scalability and adaptability for SR-Combine. It outperforms all current algorithms in both object accesses and real-time capabilities that are crucial for the acceptance of mobile services. Though the hard psychological real-time constraint of at most 3 seconds response time was not always meet, the algorithm has proven to be robust against changes of access costs and using an advanced retrieval model even allows to deliver objects successively. This is psychologically important, because users get at least some first objects within the expected response time reducing subjective waiting times.

Our future work will transfer efficient top *k* querying with *SR-Combine* into practical environments like our implementation of [WBK01]. Besides, we focus on the implementation of a mobile traffic planning service with real-time performance evaluating user preferences on travel expenses, road conditions and traffic jam awareness in a personalized manner. A first prototype is presented in [BKU02].

6. References

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

[BKU02] W-T. Balke, W. Kießling, C. Unbehend: A situation-aware mobile traffic information prototype. Tech. Report 2002-14, Institute of Computer Science, University of Augsburg, Germany, 2002 (submitted for publication)

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

[CB+00] Y.Chang, L. Bergman, V. Castelli, C. Li, M. Lo, J. Smith: The onion technique: indexing for linear optimization queries. *Intern. Conf. on Management of Data (SIGMOD'00)*, Dallas, USA, 2000.

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

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

[DR99] D. Donjerkovic and R. Ramakrishnan: Probabilistic optimization of top *k* queries. *Intern. Conf. on Very Large Databases (VLDB'99)*, Edinburgh, Great Britain, 1999.

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

[FLN01] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *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. *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. *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. *Inter. Conf. on Information Technology: Coding and Computing (ITCC'01)*, Las Vegas, USA, 2001.

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

[Kie02] W. Kießling: Foundations of preferences in database systems. *Intern. Conf. on Very Large Databases (VLDB'02)*, Hong Kong, China, 2002.

[NR99] S. Nepal and M. Ramakrishna: Query processing issues in image (multimedia) databases. *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. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, Vol. 10 (6), 1998.

[Pop97] E. Pöppel, A hierachical model of temporal perception. *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. *Compontent Database Systems*, Morgan Kaufmann, 2001.

[WBK01] M. Wagner, W-T. Balke, and W. Kießling: An XML-based Multimedia Middleware for Mobile Online Auctions. *Enterprise Information Systems III*, Kluwer, 2002.

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