ABSTRACT

Focusing on the top-K items according to a ranking criterion constitutes an important functionality in many different query answering scenarios. The idea is to read only the necessary information—mostly from secondary storage—with the ultimate goal to achieve low latency. In this work, we consider processing such top-K queries under the constraint that the result items are members of a specific set, which is provided at query time. We call this restriction a set-defined selection criterion. Set-defined selections drastically influence the pros and cons of an id-ordered index vs. a score-ordered index. We present a mathematical model that allows to decide at runtime which index to choose, leading to a combined index. To improve the latency around the break even point of the two indices, we show how to benefit from a partitioned score-ordered index and present an algorithm to create such partitions based on analyzing query logs. Further performance gains can be enjoyed using approximate top-K results, with tunable result quality. The presented approaches are evaluated using both real-world and synthetic data.

Categories and Subject Descriptors
H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval

Keywords
top-K query processing, index partitioning

1. INTRODUCTION

Reporting on ranked results is essential in many applications. Not only does it provide an ordered view on the data, according to some criteria, but it also allows users to focus on a few important results instead of inspecting hundreds or thousands of result items. Research on top-K query processing investigates how only the top portion of a hypothetically full ranking can be computed without full execution of the query—accessing only data items that are elementary for the final result (cf., [23] for an overview).

In this work, we consider set-defined selections, where the items of interest are—and whatever means—identified upfront and represented in the query as a set of item ids. In addition, the query specifies the ranking attribute of interest and the result set size referred to as parameter \( K \). Figure 1 gives an example: Attr4 is the attribute used to rank; the set \( \{2, 4\} \) restricts the result to id=2 or id=4. In case of a top-1 query with descending ordering, the result would be the tuple with id=2, which has a score of 21.6.

This problem setup is very generic and occurs in many different scenarios: Set defined selections occur frequently between independent Web services that offer different views on the same data items—like in case of Linked Open Data (LOD) portals, where different datasets capture different aspects of (the same) entities. For instance, US-born computer scientists are identified based on a knowledge base [2] and subsequently ranked based on the number of publications obtained from DBLP [13].

Our research has originally been motivated by our prior work on proposing suitable songs as soundtracks for images [53], where images are uploaded through a smartphone app. It is based on a large index containing information about one million songs, where each song is described by a numeric attribute of how good it fits to a specific sample image. A query consists of a query image and the selection set, which in this case is simply the list of songs installed on a user’s smartphone. The sample images (that correspond to attributes in Figure 1) are extracted by sampling movies. The index is 120GB big and contains information for 50 sampled movies and one million known songs—to provide enough diversity in both audio and visual dimensions. At
The query specific set \( \text{year} = 2011 \). The mentioned approaches extend the queries where the results are constrained upfront to a specific set of ids. The best performance is reached by reading the ids from the score-sorted lists.

To be able to choose the appropriate index at query time, we develop cost models for both data organizations. Still, as we will see, the latency around the break even point (the point of same performance for both indices) is the main cause of a sub-optimal performance. We show how this latency can be decreased by partitioning the data based on query logs.

1.1 Problem Formulation

Given a relation \( R \) over the attributes \( \{id, A_1, \ldots, A_N\} \), in which \( id \) is the unique identifier of an item (e.g., real world entity, document, video, image). The attributes \( A_i \) describe properties of an item and are numeric (integer or floating point numbers).

**Definition 1.** A top-\( K \) selection query is defined by the triple \((K, A_i, S)\), that is, the size of the result ranking \( K \), the attribute used for ranking \( A_i \), and a set \( S \subset \text{dom}(id) \). The task is to efficiently compute those ids that have the \( K \) largest values for attribute \( A_i \) among all ids that appear in \( R \) and the query specific set \( S \). The result is ordered by attribute \( A_i \) (w.l.o.g. in descending order).

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

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

```sql
SELECT id, A_i FROM R WHERE id IN S ORDER BY A_i LIMIT K
```

where \( A_i \) is any of the numerical attributes of \( R \). In this work, the case of a query specifying exactly one numerical attribute is considered.

1.2 Contribution and Outline

We consider the problem of efficiently processing top-K queries where the results are constrained upfront to a specified set of ids. To the best of our knowledge, this problem has not been addressed before.

In this paper, we make the following contributions:

1. We investigate the trade-offs between id- and score-ordered indices and present a cost model that allows to pick the right index at query time (Section 1).
2. We propose means to benefit from a partitioned score-ordered index and show how to create such partitions using query logs (Section 2).
3. Where approximate answers are acceptable, our algorithm can enjoy an even larger gain in latency with provided estimates for the result quality (Section 3).
4. We report on the results of a performance evaluation using real-world and synthetic data (Section 4).

2. RELATED WORK

There is a large body of existing work in the area of processing top-K queries, ranging from database systems to information retrieval. Among the most prominent approaches are the so called threshold algorithms, which aggregate scores from score-sorted lists, while maintaining a threshold used for an early termination.

Ranking join results based on aggregated scores obtained from multiple tables (i.e., top-K join processing) and embedding such ranking concepts in a query optimizer have been addressed in [22, 30]. If we ignore the ranking criterion in our setup (and, hence, want to retrieve all items), our problem consist of computing an, so called, index join (i.e., a join involving lookups in an index for at least one involved table). Then, the problem is to identify the break-even point between index lookups (e.g., a B+ tree) vs. a full table scan—for which standard textbook solutions based on cost models exist [19]. With the top-K ranking requirement, however, this “index join” has not been considered yet.

For general top-K join queries [22, 30], our studies can be of use for the direct access to the base tables in a rank-aware query plan.

Using query logs (i.e., historic workloads) with the aim of tuning a system’s performance is encountered frequently [31, 21, 35, 12]. Applications where the workload is used to improve performance vary from index defragmentation [31], over cache replacement [35] to range queries [21]. Query logs have been used in [12] with the aim to reduce the number of distributed partitions by allocating tuples that are frequently used together to the same partition. In our index organization, the graph based approach proposed in [12] is used for data partitioning.

The work in [38, 37] supports selection criteria on multi-categorical attributes (e.g., producer = ‘Ford’ and production_year=2011). The mentioned approaches extend the well known principle of data cubes for data warehousing [11] to incorporate ranking functionality. Clearly, we can not materialize data cubes for all combinations of selection sets.

Retrieving the top-K documents that contain a certain phrase can be transformed into a top-K query with range selection using data structures such as suffix trees. Data structures and algorithms for processing top-K queries with range selection, also known as top-K color queries, have been proposed in [30, 23]. It is obvious that representing set-defined selections as multiple range selections is not appropriate.

In traditional information retrieval problems, the part of the data to look at is determined by query terms, not upfront by a set of documents. A notable exception is the work by Singh et al. [32] on efficient enterprise search, where the results are restricted based on the file access rights. Knowing all access rights upfront with a clear grouping in a small number of disjunctive groups renders this problem different from the problem we address here. Another exception is the work by Bast et al. [4, 5], which uses an auto-completion search where a query is executed while a user enters it to the search box. However, the work does not consider ranking documents, hence, early termination techniques can not be applied.

3. INDICES & COST MODEL

Considering a relation \( R \) with attributes \( \{id, A_1, \ldots, A_N\} \), for each pair of attributes \( (id, A_i) \) two basic indices can be created:
The probability $P_b$ is the probability of reading one or more ids from a single block and is inverse to the probability $P_b$ that a block is not going to be read. This probability is approximated by all possible combinations of ids from all blocks without the ids from one block, $N - b$, falling into a query of size $|q|$, divided by all possible combination of all ids, $N$, falling into query of size $|q|$. This probability approximation assumes independence between ids when they are chosen into a query. This, however, does not affect the precision of the estimate as tuple ids (determining the disk position) are assigned independently of their co-occurrences in the queries (no id reassignment).

For the score-ordered index, Equation 2 is used to estimate disk blocks read ($D_b$). Here, $b$ is the number of tuples in one disk block, $K$ is the number of requested results, and $P$ is the portion of the index to be read. The probability that an id just read from disk is contained in the selection set is estimated as $\frac{|q|}{N}$, where $|q|$ is the number of ids in the selection set and $N$ is the total number of ids in the index. The expectation of a geometric distribution with this hit probability, i.e., $\frac{|q|}{N}$, gives us the portion of the index needed to be read to find one id that is contained in the selection set. As $K$ ids are needed for the final result, the portion size for one id is then multiplied by $K$. The total number of blocks is given as the ceiling of the ratio between the index portion and the block size.

$$D_b = \left[ \frac{1}{b} \times P \times K \right] ; \quad P = \frac{N}{|q|} ; \quad \Rightarrow D_b = \left[ \frac{N}{b} \times \frac{1}{|q|} \right] \times K \quad (2)$$

Once the query is submitted to the system, the execution-time estimates are calculated and the index with the smaller execution time is used to answer the query. This procedure is totally hidden from the application layer: it appears as one index that combines the best of two index organizations. We call this index combined index. With an ideal cost model at hand, the combined index has a performance as illustrated in Figure 2.

### 4. Partitioned Index Organization

The above model provides a solid mechanism to identify the best index to use. In fact, experiments show that it is (almost) perfect. On the other hand, the characteristics of the underlying indices lead to a degenerated performance of the combined index around the break-even point (cf., Figure 2).

The above cost model reveals a way to improve runtime beyond the point of simply choosing the right index: The index size in terms of number of tuples stored, has a strong influence on the query response time, while the query features themselves can not be influenced.

Thus, the main idea of a partitioned index is to organize the original index into multiple chunks, such that a large fraction of queries is answered by reading only from one of them. This has a high potential: partitioning the score-ordered index into $m$ parts lowers the query answering time by a factor of $m$ (the number of blocks read from the disk would be $m$ times smaller).

The large score-ordered index is chopped up in a set of non-overlapping partitions. Each partition is organized as a score-ordered index. The decision which tuples to put together in a partition is done using a graph-based clustering approach.

To fully harness such a partitioning, we check at query time whether the selection set is

---

**Figure 2:** Characteristic performances of id-ordered index, score-ordered index, and the ideal combined index.

- an index on the id attribute (called **id-ordered index**)
- an index on the numerical attribute $A_i$ (called **score-ordered index**)

The id attribute are assumed to be densely populated in sequential order, such that the position of a score on the disk can be calculated directly from the id value. Hence, only scores need to be stored—not the ids themselves. If the ids are not sequential, existing techniques based on B+ trees (cf., e.g., [19]) can be used for indexing. We adapt a column-store data layout where the relation $R$ is stored on disk in a per-attribute fashion (not row-by-row).

Both index organizations come with advantages and disadvantages: The id-ordered index is ideal if the size of the query set is rather small, resulting in a small number of index lookups. In contrast, the score-ordered index is ideal if the size of the query set is large, such that $K$ items out of the query set are found very early when scanning the sorted list on disk.

To benefit from both sweet spots at the same time, a cost model is required to decide at query time which index to use. We optimize for low query response time, which is modeled as

$$t = c_1 + c_2 \times D_b$$

where $D_b$ is the size of the data read from disk and $c_1$ and $c_2$ are constants which minimize the squared error on real-world measurements. The intuition is that $c_2$ represents the data transfer time, while $c_1$ approximates the time needed for a random access to disk. To keep the analysis tractable, we ignore influences of distributions of ids and scores, and, hence, treat the size of the selection set and the value of $K$ (specified in the query) as the main ingredients for the amount of data read from disk. The latter is represented in number of disk blocks, as the access to disk is naturally done in a block-based manner.

For the id-ordered index, Equation 4 is used to estimate the number of disk blocks read, $D_b$. $N$ represents the total number of tuples in the index, $b$ is the number of tuples stored in a single disk block, and $P_b$ is the probability that a single block is going to be read.

$$D_b = \frac{N}{b} P_b ; \quad P_b = 1 - \overline{P}_b ; \quad \overline{P}_b = \left( \frac{N - b}{N} \right) \left( \frac{|q|}{|q|} \right) \Rightarrow D_b = \frac{N}{b} \left( 1 - \frac{(N - b)}{(N \times |q|)} \right) \quad (1)$$
in one partition,

ii) mostly contained in one partition,

iii) distributed between many partitions.

Answering a query in the first case is done using only the selected partition, while the second case requires a lookup of missing tuples using the id-ordered index. In the third case the query is answered using the combined index.

To determine if the partitioned index should be used for query answering, we employ the cost models introduced in Section 6. If one partition captures the entire selection set, only the model for the partitioned index is used, which essentially is the model for a score-ordered index, where the index size is adjusted accordingly.

In case not all of the ids from the selection set are found in one partition, the intersection size is used to estimate the query response time of the partitioned index. The number of the remaining ids (which are not covered by the partition) is used to estimate the lookup cost of the scores in the id-ordered index. The sum of the two estimates is used as a final response time estimate in case of this mixed access to the partitioned and the id-ordered index.

4.1 The Partition Selection Phase

To determine the partition which contains the largest subset of the selection set, data structures for computing set intersections are required. A bit-set structure is created for each partition, where each id is represented by one bit. The bit indicates whether or not the id is contained in the partition. This representation is exact (no false positives, no false negatives).

First, we consider the case when these bit set structures are kept in main memory, and then we focus on the case when they are stored on the disk with compact sketches used to decide which partition to access on disk.

Bits Sets in Main Memory

Bit sets in main memory allow a fast calculation of the intersection between the query selection set and each of the partitions in the index. *Only the partition with the largest intersection is used*, in case the time estimate using this partition is less than the time estimate for the combined index. Otherwise the query is answered using the combined index. This cost assessment is easy to achieve as the available bit sets give precise (exact) numbers of the contained and missing ids in a partition.

Bits Sets on Disk

If the bit sets do not fit in main memory, reading them from disk would in most cases consume more time than answering a query using the combined index. Our solution to this problem keeps only compact sketches \[\text{(K-Minimum Values)}\] of partitions in main memory and the full sketch information on disk in the header of the corresponding partition. At query time, these sketches are used to determine the most promising partition to access by estimating the intersection size between partitions and the query selection sets. The accuracy of this estimation can be tuned using sketch specific parameters. In this work, KMV sketches \[\text{(k-Minimum Values)}\] are used.

The structure for the partitioned index with sketches is shown in Figure 3. The bit set structure for a given partition is located on the disk at the beginning of the partition itself. This is done such that once the bit set is read—and it tells that the partition is useful—the reading of the actual partition content can continue without an additional disk seek.

### Partition 1

<table>
<thead>
<tr>
<th>01001</th>
<th>418752</th>
<th>10234</th>
</tr>
</thead>
</table>

### Partition 2

<table>
<thead>
<tr>
<th>10110</th>
<th>14456</th>
<th>13238</th>
<th>9584</th>
</tr>
</thead>
</table>

Figure 3: Partitioned index structure with sketches

Having only rough sketches of the partition contents in main memory requires changes to the querying algorithm. First, the most promising partition is identified using sketches. Then, the bit set for this partition is read and used to calculate the exact intersection size between the partition and the selection set. This information is then used to estimate if the selected partition is beneficial for answering the query.

It is important to avoid accessing a partition on disk that turns out to be of little use once the bit set is inspected. To limit these wrong decisions, made by the estimation inaccuracy of the sketches, a partition is identified as promising only if we are highly confident that it will be useful for the query optimization later on. The problem with sketches is that comparing a huge list of ids with a relatively small selection set often results in an empty intersection. We identify a partition to be useful, if and only if the intersection with the selection set is estimated to be larger than zero and there is no other partition for which the estimate is non zero.

In case partitioned indices are created for multiple attribute pairs of the same relation (e.g., \((id, A_1), (id, A_2), \ldots\)), the content of the partitions will be the same for all of them. Hence, bit sets describing the partitions are materialized only once—they are shared to reduce the overhead.

4.2 Index Partitioning

The problem of data partitioning is formulated as follows: given selection sets from a query log, create \(m\) disjoint data partitions such that the probability of finding a randomly selected pair of ids from a randomly selected selection set in a single partition is maximized. The partitions should further be approximately equal in size. Determining the optimal number of partitions is not trivial: A large number of ideal partitions (i.e., each selection set is completely found in one partition) would decrease the runtime. However, increasing the number of partitions would increase the error introduced by the partitioning, that means, less and less queries could be answered by a single partition. In this work, we determine the optimal number of partitions experimentally (cf., Section 6).

For partitioning, we employ a technique used in recent work on data partitioning in distributed database systems, by Curino et al. [12]. The basic idea of their approach, coined Schism, is to create a graph based on a database workload (query logs). The vertices of the graph are tuples with edges connecting frequently co-occurring tuples in the transactions. The edge weight is given by the number of transac-
The graph, where \( q \) is the size of the query selection set. To avoid a quadratic explosion, in particular for queries with two partitions: (1, 4) and (2, 3, 5). This way, each query contributes \( \frac{(|q|-1)\cdot(|q|-2)}{2} \) edges to the graph, where \( |q| \) is the size of the query selection set. To avoid a quadratic explosion, in particular for queries with larger selection sets, we introduced edge sampling. For each of the ids in the given selection set, 10 randomly selected ids from the same set are used to create edges between them, rather than using all of the other ids. This way we get \( 10 \cdot |q| \) edges for each query, while preserving the density in the graph.

The basic assumption behind index partitioning is that selection sets are clustered in a meaningful way. Although this might not hold in general, selection sets are usually coherent in a semantic way (e.g., most people possess songs from mainly one genre).

Constrained k-way graph partitioning is NP-complete, but there exist efficient and accurate approximation techniques. In this work, we are using the software package METIS [25, 28], a freely available software library that offers approximate constrained k-way partitioning based on multilevel coarsening techniques.

### 5. APPROXIMATE QUERY ANSWERING

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

#### 5.1 Tunable Expected Precision

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

In the following, we derive an estimate for the results quality provided by the approximate query answering algorithm. Using this estimate at query time enables us to fall back to the exact query answering mode, in case this expected result quality is below the specified minimum threshold.

We describe the probability distribution of the precision as a binomial distribution with “success probability” \( p_h \). This \( p_h \) is the probability that an id is found in the partition, given that it is in the selection set and also a true top-K element. Note that, due to the nature of the partitions and the retrieval algorithm, \( p_h \) is the same as the probability that a retrieved top-K element is a true top-K element.

The estimated precision is then given by the expectation of the binomial distribution, divided by the size of the results,

\[
P(X \leq x) = \sum_{i=0}^{\lfloor x \rfloor} \binom{K}{i} p_h^i (1 - p_h)^{K-i}
\]

\[
prec = \frac{E[X]}{K} = \frac{p_h \times K}{K} = p_h
\]

where \( K \) is the size of the results (\( K \) in top-K) and hit probability is \( p_h \).

The success (or hit) probability \( p_h \) can be written as

\[
p_h = P(id \in F | id \in S)
\]

with \( F \) representing the partition and \( S \) representing the selection set. Applying Bayes’ theorem and assuming conditional independence between an id being found in the top-K results and the id being found in partition, given that it is in the selection set, we can derive

\[
p_h = \frac{P(id \in F | id \in S) \cdot P(id \in S)}{P(id \in S)}
\]

Applying the definition of conditional probability and using

\[
P(X, Y) = P(X)P(Y) + \text{cov}(X, Y)
\]

for Bernoulli random variables, we obtain:

\[
p_h = \frac{P(id \in F | id \in S)P(id \in S) + \text{cov}(id \in F, id \in S)}{P(id \in S)}
\]

Estimating that \( P(id \in F) = \frac{|F|}{N} \) and that \( P(id \in S) = \frac{|S|}{N} \) (where \( N \) is the total number of tuples), together with the covariance estimation \( \text{cov}(id \in F, id \in S) = \frac{|F \cap S|}{N^2} - \frac{|F| \cdot |S|}{N^2} \), gives us:

\[
p_h = \frac{\frac{|F| \cdot |S|}{N^2} + \frac{|F \cap S|}{N^2} - \frac{|F| \cdot |S|}{N^2}}{\frac{|S|}{N}} = \frac{|F \cap S|}{|S|}
\]

This shows that the precision is estimated as the size of intersection between the selection set and partition, divided by the selection set size. For instance, if there are 90% of the selection set contained in the queried partition, we estimate the precision to be 90%.

To get an accurate precision estimate, we use the exact intersection size using the partition bit sets that are read from the disk once the partition is selected based on the sketches.

### 6. EXPERIMENTAL EVALUATION

We implemented the described indexing techniques and algorithms in Java 1.6 using direct access to the disk by a JNI (Java Native Interface) connection to routines written in C. The direct IO access uses the \_DIRECT flag in the
libraries provided by the operating system. One disk block is kept cached in main memory while data is read from the disk. As a direct access to the disk is used, the size and the speed of the main memory was of negligible influence to our measurements.

All experiments are conducted on a Linux machine (Ubuntu 11.04 64bit with kernel version 2.6.32-28) with an Intel Xeon W3530 CPU (2.8GHz, 8MB cache, 4 cores (8 threads)). The local disk used is a 3Gb/s Barracuda 7200.12 SATA (7200 rpm, 32 MB cache, avg. latency 4.16 ms, random read seek time: 8.5 ms). The routines are executed using Java SE runtime version 1.6 64-bit, with C routines compiled using gcc version 4.4.3.

**Prerequisites:** To estimate the query response time, the cost models (Section 8) need to know the block size. This, however, depends on how much the disk is actually reading ahead with each access to the physical disk. In our case, we measured a size of 896 KB.

**Real-world Dataset**

As for the real-world dataset we get back to the problem that originally motivated our research. With Picaso, we have developed an approach to suggest songs, out of a user provided collection, to pictures submitted by a user to our system. We analyzed the Million Song Dataset [6], which contains low-level feature descriptors of one million songs. This way we need only the title of a user’s mp3 and can recommend songs out of this set, for a given picture. A user provided set of songs is interpreted as a selection set.

In addition, 50 movies are processed and analyzed resulting in approximately 10,000 movie snapshots. Fitness information between each song and each movie snapshot is calculate and stored. With 4 bytes for an id and 4 bytes for the score encoding, the total size of the indexed information is around 120GB for both id-ordered and score-ordered indices.

We extended the 1 million songs to 14 million songs to account for a more realistic total count of songs in the world. This approximation is based on the overlap between the user profiles (i.e., sets of songs in users’ collections) we received and the list of songs in the free base [13] database. The scores for those artificially added songs are generated from the distribution of scores for the songs in the million songs dataset.

For one index entry, the score in case of the id-ordered index and both the score and id in case of the score-ordered index are stored—which sums up to 12 bytes in total. As one tuple needs one bit in the partition bit set, the size of the bit set for one partition is 12 × 8 (= 96) smaller than the combined index size of one column (e.g., movie snapshot). As bit sets are the same for all columns, thus, the required space for the bit set information of one partition is a factor of 960,000 smaller than the whole combined index (relation). When the bit sets are stored on disk, we used 28,000 minimum values in the KVM-sketch (each value is of 4 bytes size) for each partition, rendering the signature for one partition 60,000 times smaller than the combined index size of one column. As an illustration: with only 1 GB of main memory used and 10 partitions one can index 5.85 TB of data on disk using sketches.

**Queries:** We use 150 user profiles, i.e., sets of songs, obtained through our iPhone app PicasSound. The profiles are anonymized and do not contain any personal information. Each of those profiles represents one selection set that we use as the top-10 query in this study. We execute each query ten times and the average runtime is reported for that query. Each execution is accessing disk (no caching), which is a direct consequence of direct IO usage. The distribution of the selection set sizes for the iPhone queries is shown in Figure 5.

**Partitioning:** as 150 queries obtained from iPhone users are not enough for both testing and partitioning purposes and in particular to also show that the partitioning can be done on different but semantically related data, we use publicly available data [10] from last.fm [26] for the partitioning. This dataset contains the most-listened-to artists for 360,000 last.fm users. Although this dataset contains artists rather than individual songs, we partition the artists and then assign songs to the partition which contains the corresponding artist. The partitioning of the artists is done as described in Section 4.2 where the top artists of one user are considered as a query used in graph creation.

As the partitioning dataset and the query dataset have different sources, we measured that around 82% of the artists present in the queries are also contained in the partitioning dataset. We removed song ids from the queries’ selection sets if they were not contained in the partitioning set.

**Synthetic Data Set**

For the synthetic dataset, we use the same song collection with the same scores used in the real-world dataset, but create synthetic queries for execution and queries used in the partitioning process. This allows varying the query clustering density to study its effect on the performance of the partitioned index. To create the artificial clustering, ids are randomly split into 10 groups (artificial clusters). Then, we generate queries based on these groups.

We distinguish three types of queries: (i) all ids from the selection set are found in one cluster (ii) ids are split between two clusters and (iii) ids are uniformly spread between all clusters. Based on the ratio between these types of queries, we create three query logs for partitioning and testing purposes. The first query log, named “best” contains 80% of type (i) queries, 10% of type (ii) queries, and 10% of type (iii) queries. The second query log, named “middle”, contains 60% type (i), 20% type (ii), and 20% of type (iii) queries. Finally, the third, named “worst” query log contains 40% type (i), 30% type (ii), and 30% of type (iii) queries.

**Queries:** 200 queries for testing purposes were generated for each of the query log types with selection size uniformly distributed between 10 and 1000. Each query is generated by randomly selecting ids from one, two, or all artificial clusters. For instance, the “best” query log has 80% queries for which ids are selected from one randomly selected artificial cluster.

**Partitioning:** 20,000 queries for each of the query log
types are generated, with the selection size uniformly distributed between 10 and 500.

6.1 Model Verification

First, we experimentally evaluate the accuracy of the runtime estimates obtained by the models proposed in Section 6.2. We use the songs dataset and create top-20 queries with uniformly selected ids in the selection set.

First, we measure the effect of the table size (in terms of number of tuples) on the runtime of the score-ordered index. The results are shown in Figure 8(a): we distinguish three index sizes, 14 million tuples, 7 million tuples, and 3.5 million tuples. We can see that the runtime drops almost by a factor of two when decreasing the index by a factor of two. This shows that the runtime is proportional to the number of tuples stored in the index as proposed by the model (cf., Equation 2). The effects of changing $K$ on the runtime using the score-ordered index is shown in Figure 8(b). We observe that a larger value of $K$ increases the runtime proportionally, which is again in accordance to the model. After fitting the $c_1$ and $c_2$ parameters, we see that the estimated values are almost a perfect fit to the measured values, see Figure 8(d).

The effects of the table size on id-ordered index are shown in Figure 8(a), where we see that the runtime is proportional to the table size. This plot also reveals small anomalies for small query sizes, in which case the 7 million songs index is performing better than the 3.5 million songs index. This together with the sudden decrease in runtime for 14 million songs at query size 200 is most likely the effect of the disk’s adaptive read-ahead strategy. Figure 8(c) shows that the values estimated the model are very close to the measured ones.

Figure 8(b) shows that the parameter $K$ has no effect on the runtime for id-ordered index. This is also in accordance to our model (cf., Equation 1).

The quartiles of multiple runtime measurements on the score-ordered index (Figure 8(c)) and the id-ordered index (Figure 8(c)) are shown for 14 million songs and $K=20$. We see that the runtimes have low variance, which means that time estimates are good in general, not only on average.

Overall, we see that the proposed models give a good estimate of the runtime for both indices. It is important to note that they do not have to be perfect in estimating the absolute runtimes, rather we need them to provide a reasonable decision of which index is better suitable to answer a query. To get a better understanding of this, we study the performance of the combined index which highly depends on the runtime estimates. The results for the real-world dataset are shown in Figure 8. We observe that the combined index performs almost optimally, choosing the right index for almost all of the queries, with only a couple of misses around the break-even point where the difference in runtime of the id- and score-ordered indices is negligible.

6.2 Exact Top-K Retrieval

We report on the results for the partitioned index evaluation, when the index is used to retrieve the exact top-K results. We measure the runtime of the partitioned index on the real-world dataset with 10 partitions, when the bit sets are in main memory (Figure 10(a)) and when they are on disk (Figure 10(b)). The selection set size is shown on the x-axis, while the y-axis shows the runtime in milliseconds.

We summarize these results in Figure 10, where query sizes are grouped into ranges of size 50. For a better readabil-

![Figure 6: Runtime combined index](image)

![Figure 7: Partitioned index runtime (a) bitsets in main memory (b) bitsets on disk](image)
dataset and get smaller with the middle dataset and are smallest for the best synthetic dataset. We can see that the gain for the worst dataset, when the bit sets are kept on disk, is tiny, almost nonexistent. However, when the bit sets are kept in main memory, there is a gain for even the worst setup. Comparison between keeping bit sets in main memory and on disk is shown in Figure 13 (a) for best synthetic dataset and in Figure 13 (b) for worst synthetic dataset.

### 6.3 Approximate Top-K Retrieval

We have measured the runtime for the approximate querying, with and without guarantees, using the real-world dataset, shown in Figure 14. We see that the average runtime for the approximate querying without guarantees is much lower than for the variant with guarantees, e.g., for query sizes between 100 and 150 average runtime without guarantees is 83 milliseconds while runtime with guarantees is 140 milliseconds. This of course comes with the cost of decreased in precision. Precision of 80% for querying with guarantees in this case, while runtime without guarantees is 83 milliseconds while runtime with guarantees is 140 milliseconds.

Varying the requested (minimum) precision from 60% to 90% for approximate querying without guarantees is much lower than for the variant with guarantees, e.g., for query sizes between 100 and 150 average runtime without guarantees is 83 milliseconds while runtime with guarantees is 140 milliseconds. This of course comes with the cost of decreased in precision shown in the same Figure 14. We used a requested precision of 80% for querying with guarantees in this case, which resulted in average precision around 95%.

Varying the requested (minimum) precision from 60% to 90% for approximate querying with guarantees is shown in Figure 15. We see that the precision drops down when the requested precision is lowered. Lowering the requested precision results also in lower runtimes, as reported in Figure 15.
The results of the approximate querying without guarantees over synthetic data are shown in Figure 16. As expected, the runtime for best synthetic dataset is lowest, the middle synthetic dataset has medium runtime, and the worst setup incurs the highest runtime. We see that even for the worst synthetic dataset, the improvement in runtime over the combined index is still high (up to 105 ms). As the modeling in three cases has little control over the precision, it varies independently of the synthetic dataset use (Figure 16).

Approximate querying with guarantees and a requested precision of 90% over synthetic data results in an average precision close to 100%, independently of the dataset used, cf., Figure 17. This emphasizes once more the influence of the requested precision over the actually observed result precision. The measured runtime is again as expected. It is lowest for the best synthetic dataset, highest for the worst dataset, and medium for the middle one, cf., Figure 17.

7. CONCLUSION

In this work, we addressed the problem of finding the top-K items out of a global index, where the choice of result items is restricted to a query-dependent subset. This problem is very fundamental: it appears in cases of distributed services on the Web (e.g., linked open data) or at the source layer of rank-aware query processing in databases, etc. To our knowledge, we are the first to consider this problem, which is completely different from common top-K aggregation queries in the literature. Our approach is based on a careful analysis of the pros and cons of an id- vs. a score-ordered index. We derived a cost model to choose the most suitable of these indices at runtime. We devised a way to benefit from a partitioned data organization and investigated the influences of an approximate query answering. The conducted performance evaluation, based on real-world as well as synthetically generated data, revealed that the cost model is (almost) perfect in deciding which index to choose. While the combined index is, hence, almost optimal, we saw that based on data partitioning, the performance around the break-even point can still be drastically improved. We further saw that the derived estimate for the result quality—in case of approximate query answering—provides a robust and intuitive mechanism to trade off result quality for runtime performance.

8. REFERENCES

Figure 15: Approximate querying with guarantees, real-world dataset

Figure 16: Approximate querying with guarantees, synthetic dataset

Figure 17: Approximate querying without guarantees, synthetic dataset


