

Progressive and Selective Merge: Computing Top-K with Ad-hoc Ranking Functions*

Dong Xin Jiawei Han Kevin Chen-Chuan Chang
Department of Computer Science
University of Illinois at Urbana-Champaign
{dongxin, hanj, kcchang}@uiuc.edu

ABSTRACT

The family of *threshold algorithm* (i.e., TA) has been widely studied for efficiently computing top- k queries. TA uses a *sort-merge* framework that assumes data lists are pre-sorted, and the ranking functions are *monotone*. However, in many database applications, attribute values are indexed by tree-structured indices (e.g., *B-tree*, *R-tree*), and the ranking functions are not necessarily monotone. To answer top- k queries with *ad-hoc* ranking functions, this paper studies an *index-merge* paradigm that performs progressive search over the space of joint states composed by multiple index nodes.

We address two challenges for efficient query processing. First, to minimize the search complexity, we present a double-heap algorithm which supports not only progressive state search but also progressive state generation. Second, to avoid unnecessary disk access, we characterize a type of “empty-state” that does not contribute to the final results, and propose a new materialization model, *join-signature*, to prune empty-states. Our performance study shows that the proposed method achieves one order of magnitude speed-up over baseline solutions.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems - Query processing

General Terms: Algorithms

Keywords: Top- k Query, Progressive Merge, Selective Merge

1. INTRODUCTION

Top- k queries ask for k tuples ordered according to a specific ranking function that combines the values from multiple attributes. Efficient computation of top- k results has been one of the focusing points in research with numerous studies reported. A widely studied implementation method is the family of threshold algorithms (i.e., TA) [10, 12, 11]. TA uses a *sort-merge* framework that sequentially scans a set of pre-computed lists, each of which is sorted according to in-

dividual scores, and merges data objects from different lists to compute the aggregated scores. Usually, it stops scanning long before reaching the end of the lists. TA has been extensively studied in the context of middle-ware, information retrieval, and multimedia similarity search, where the aggregation functions are usually *monotone*. We say that a function f is monotone if $f(x_1, x_2, \dots, x_m) \leq f(x'_1, x'_2, \dots, x'_m)$ whenever $x_i \leq x'_i$, for every i . However, in many database applications, the ranking functions are not monotone, as demonstrated in the following examples.

Example 1. In many business applications, prediction models, which are trained by historical data sets, are used to predict a target value based on other attributes. As a case study, suppose that someone wants to predict stock value V , based on earnings E and research expenditure R . A prediction model $f(E, R)$ captures the trade-off between the current earnings E and future benefits by R . A typical query over the stock database is to find the top- k stocks that best match the prediction model.

select top- k s .CompanyName from Stock s
order by $(s.V - f(s.E, s.R))^2$ asc ■

Example 2. Consider a house database that maintains the *price* and *location* (e.g., *longitude* and *latitude*) of each house. To search for a house that has expected price P and also is close to a good school S (using Manhattan or Euclidean distance d), we may formulate the ranking function as below, where α is a weighting parameter.

select top- k h .Address from House h
order by $\alpha|h.price - P| + d(S, h)$ asc ■

In both above examples, the ranking functions are not monotone, and thus the sort-merge framework is not applicable. The naive solution that scans the whole database is not desirable for a large database and small k values. On the other hand, database systems often organize data attributes using tree-structured indices (e.g., *B-tree*, *R-tree*), and the attributes involved in the ranking function are generally distributed in multiple indices. For example, in Ex. 1, V , E and R are typically indexed by three different *B-trees*, and in Ex. 2, *price* and *location* may be indexed by a *B-tree* and an *R-tree*.

For a relatively small value of k , the processing time can be reduced significantly by progressively merging indices. In general, indices are accessed in a top-down fashion such that the roots are first retrieved and child nodes are visited after their parents. To optimize the disk access, an important issue is to determine the right order to retrieve

*Work supported by the U.S. National Science Foundation NSF IIS-03-08215/05-13678/01-33199 and BDI-05-15813.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'07, June 12–14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...\$5.00.

nodes (i.e., *access scheduling*). Under the context of index merging, this scheduling problem often involves search over the space of joint states composed by multiple index nodes. We refer to this procedure as *state search*. Following Ex. 2, suppose *price* and *location* are indexed by a *B*-tree *P* and an *R*-tree *L*. The index-merge starts with the root state (*P.root*, *L.root*), and is recursively called on the next promising state, until the top-*k* results are found. To determine the next state to access, the algorithm takes the Cartesian product of all child nodes of *P.root* and those of *L.root*, and each pair is a new joint state. We refer to this procedure as *state generation*.

In this paper, we study the problem of *efficient processing top-k queries with ad-hoc ranking functions by an index-merge paradigm*, thus extending TA’s sort-merge framework for monotone aggregate functions. While in sort-merge, data is accessed sequentially along the sorted lists, in index-merge, access scheduling is more complicated (e.g., by state search and state generation). We address two challenges for the index-merge framework. First, towards CPU optimality, how to minimize the number of states to be generated? Secondly, towards I/O optimality, how to prune unnecessary disk accesses? To demonstrate the significance of these two challenges, Table 1 shows a performance comparison between the basic index-merge and the improved algorithm developed in this paper ¹.

Index-Merge	States Generated	Disk Accesses
Basic	420, 323	4, 133
Improved	9, 237	483

Table 1: Significance of the two challenges

For the first challenge, similar work to reduce the search complexity was previously studied by spatial (or distance) joins [6, 19, 23], which ask for pairs of objects intersecting (or close) to each other. Two popular optimization techniques were proposed to reduce the complexity of joint space search by joining two *R*-trees. The first, search space restriction, performs two linear scans over the entries of both nodes, and prunes out of each node the entries that do not intersect with the other node. The second, plane sweep, applies sorting on one dimension in order to reduce the cost of computing entry pairs to be joined. Unfortunately, all these methods are tailored for spatial functions that measure intersection or distance between spatial objects. There is no principled solution for ad-hoc ranking functions.

The second challenge is also imposed on the sort-merge approach in that objects near the top on one list may rank low in other lists, and hence a large portion of the retrieved data may not contribute to the final results at all. To remedy this, random access [1, 7, 8, 12, 11, 16] is used to resolve missing scores of partially merged objects. In this paper, we show that random access, although effective in sort-merge, does not lead to early termination in the index-merge framework (see Section 3 for detailed descriptions), and thus the previously studies are not applicable.

This paper proposes new strategies for both challenges. To reduce the search complexity, we decouple the state search and state generation procedures. Instead of fully expanding

¹The results are collected on a top-100 query with ranking function $f = (A - B^2)^2$, which merges two *B*₊-tree indices on attributes *A* and *B*. The database has 1*M* tuples.

a joint state during search, our algorithm progressively and partially expands a state when necessary. We refer to this strategy as *progressive merge*. To prune unnecessary disk access, we characterize a type of “empty-state” that does not contribute to the final results, and present a simple materialization model, *join-signature*, to prune empty states. We refer to this strategy as *selective merge*. Our experimental results show that the new method achieves one order of magnitude speedup over the basic index-merge approach. More specifically, the contributions of this paper are:

- We conduct a thorough analysis of the index-merge framework and study two optimal scenarios for query processing: (1) CPU optimal, which touches the minimal number of states in search; and on top of that, (2) I/O optimal, which retrieves the minimal number of index nodes.
- We propose a *double-heap* algorithm which decouples the state search and state generation procedures. For all joint states scheduled for access, a global heap facilitates search; and for each joined state in the global heap, a local heap controls the progressive expansion.
- We develop a *threshold expansion* method to progressively expand a joint state for general functions. A function, though general over the entire region, may be monotone (or semi-monotone) in some sub-regions. A *neighborhood expansion* strategy can thus be developed.
- We propose *join-signature* to effectively prune disk access. The join-signature is a compressed summarization that indicates whether a joint state is empty or not. We show that the join-signature is compact, easy to compute and incurs very low overhead during query processing.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the framework for query processing, and analyzes the problem. The double-heap algorithm and state expansion strategies are developed in Section 4, and the selective merge with join-signature is presented in Section 5. We report the experimental results in Section 6, discuss the extensions in Section 7, and conclude the study in Section 8.

2. RELATED WORK

We have addressed the TA algorithms and the spatial joins in the last section. Here, we discuss other related work.

Our index-merge uses a top-down search paradigm. Computing top-*k* by interleaf traversal, which starts search from the leaf-nodes containing extreme points and progressively traverses to neighboring leaf-nodes, was exploited by [21, 22]. The method requires the neighboring leaf-nodes to be well defined, and this may not be available in some index structures (e.g., *R*-tree). On the other hand, top-down search is more general. Moreover, we develop an adaptive search strategy for the top-down framework such that it conducts neighborhood expansion whenever applicable and switches to threshold expansion otherwise (see Section 4).

The join-signature proposed in this paper is an extension of data cube [9] which pre-computes multi-dimensional aggregates. We treat each index as a dimension, and index nodes as values. With a boolean measure that indicates whether a joint state is empty or not, a join-signature is essentially a data cube over multiple index-dimensions. Materializing join results is also explored by join indices [20] and ranking cube [21]. The join-signature differs from them

tid	A	B	$f = (A - B)^2$
t1	10	40	900
t2	20	60	1600
t3	30	65	1225
t4	50	45	25
t5	54	10	1936
t6	72	30	1764
t7	75	36	1521
t8	85	62	529

Table 2: A Sample Database

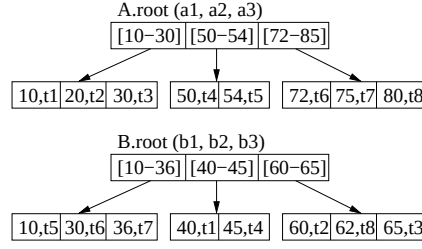


Figure 1: Indices on A and B

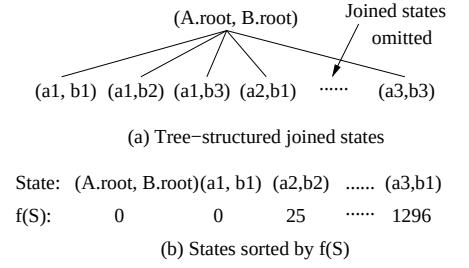


Figure 2: Space of Joint States

in that the join-signature is built at the index node granularity, rather than on the data record level.

Top- k queries are related to several other preference queries, including skyline query [5] and convex hull query [4]. The methodology developed in this paper is also applicable to these queries under the context of index-merge, and we will discuss the extensions in Section 7.

3. PROBLEM ANALYSIS

3.1 Query and Data Model

The task of finding top- k tuples from a database can be posed with either a maximization or a minimization criterion. Since a maximal query can be turned into a minimal one by switching the sign of objective function, in the rest of the paper we assume *minimization queries* are issued. Given a relation R with attributes $\mathcal{A} = \{A_1, A_2, \dots, A_n\}$, a top- k query with an evaluation function f , which is formulated on a subset of attributes $\{A'_1, A'_2, \dots, A'_m\} \subseteq \mathcal{A}$, asks for k data tuples $t_1, t_2, \dots, t_k$ such that for any other $t \in R$, $f(t) \geq \max_{i=1}^k f(t_i)$. We further assume that the ranking function f has the following property: *Given a function f and the domain region Ω on its variables, the lower bound of f over Ω can be derived.*

We assume attributes involved in f are indexed by some hierarchical indices (e.g., B -tree or R -tree), such that a subspace occupied by a tree node is always contained in the subspace of its parent node. Suppose indices $\{I_1, I_2, \dots, I_m\}$ are used to answer a top- k query. The space of joint states inherits the hierarchical property of the index. More specifically, the joint state can be recursively defined as follows. The root state is $(I_1.root, I_2.root, \dots, I_m.root)$, and for each joint state $(I_1.n_1, I_2.n_2, \dots, I_m.n_m)$, its child states are the Cartesian products of child nodes of $I_i.n_i$ ($i = 1, \dots, m$). If $I_i.n_i$ for some $i = 1, \dots, m$ is a leaf node, $I_i.n_i$ itself is used in the Cartesian products to generate child states. If all $I_i.n_i$ are leaf nodes, the joined state is a leaf state. Table 2 shows an example database, which consists of two attributes A and B (f is a demonstrating ranking function). Indices on A and B are shown in Figure 1 and the joint state space is assembled in Figure 2.a.

3.2 Framework for Query Processing

To compare with, we review the *sort-merge* algorithm used by TA. TA sequentially retrieves data from the sorted lists. At any state, we can classify tuples into three categories: fully merged, partially merged and unseen. The algorithm maintains an upper bound² for the current top- k fully merged tuples and a lower bound for the partially

²Note we use minimization queries in this paper.

merged and unseen objects. If the upper bound score is no larger than the lower bound score, TA halts and returns the top- k results. Additionally, for those partially merged objects, unknown scores from some attributes can be looked up by random access, thus facilitating the early termination.

Similarly, in *index-merge*, we need to define two strategies: (1) a way to schedule node access, and (2) a stop condition that guarantees the top- k results are found. Given the value boundaries in index nodes, each joint state S is associated with a domain region $\Omega(S)$. We define $f(S)$ as the lower bound of the ranking function f over a joint space S . For example, in Figure 2, $\Omega(a1, b1) = [10, 30] \times [10, 36]$, and $f(S) = 0$ for ranking function $f = (A - B)^2$. Clearly, for each joint state, $f(S) \geq f(\text{parent}(S))$. The index-merge method starts with the joint root state, progressively finds the state with minimal $f(S)$ and examines its child states. In the meanwhile, the algorithm maintains an upper bound score of the current top- k results. The search stops when the upper bound score is no larger than the minimal $f(S)$ (for all rest S). The procedure is described in Algorithm 1.

Algorithm 1 Query Processing by Index-merge

Input: A set of Indices I , ranking function f , top- k

- 1: $TopK = \phi$; //heap with size k to hold current top- k
 - 2: $g_heap = \{Joint\ Root\}$; //heap for state search
 - 3: **while** ($g_heap \neq \phi$ and $f(TopK.root) > f(g_heap.root)$)
 - 4: remove top entry S from g_heap ;
 - 5: **if** S is a leaf state
 - 6: Retrieve data and update $TopK$;
 - 7: **else** // S is a non-leaf joint state
 - 8: insert all child states of S to g_heap ;
 - 9: **return**
-

We briefly explain the query processing as follows. The algorithm maintains two heaps. The *TopK* heap keeps current best k results that are fully merged, and the root of the heap is the k^{th} tuple. The *g_heap* maintains candidate states for search, and the state with minimal $f(S)$ appears at the root. We also maintain a hashtable h for tuple merge. Whenever a leaf state is retrieved, we look up tuples in h . If a tuple t is fully merged, and is better than $TopK.root$, we remove the root from *TopK* and insert t to *TopK*.

3.3 Optimal Access Scheduling

We address two types of optimality for access scheduling: type-I that is CPU optimal and type-II that is disk-access optimal. By bridging Algorithm 1 and the optimal cases, we then identify two challenges.

3.3.1 Type-I Optimality

Suppose the upper bound score of final top- k results is s^* (remember that we use minimal k criteria in this paper). *The type-I optimal scenario is that the algorithm only enumerates state S such that $f(S) \leq s^*$.* We denote the optimal number of states as $n_I^* = |\{S | f(S) \leq s^*\}|$. As an example, the top-1 query with function $f = (A - B)^2$ on the sample database (Table 2) returns t_4 as the final result. $s^* = 25$, and only $(A.root, B.root)$, $(a1, b1)$ and $(a2, b2)$ need to be enumerated. The states generated by Algorithm 1 can be classified into: (1) *examined states* that appear on Line 4; and (2) *generated candidates* that appear on Line 8. We have the following lemma.

LEMMA 1. For a top- k query involving m indices, the number of examined states by Algorithm 1 is n_I^* , and the number of generated candidates by Algorithm 1 is upper bounded by $n_I^* \prod_{i=1}^m M_i$, where n_I^* is the optimal number of states given by type-I optimality, and M_i is the node fanout of index i .

PROOF. We prove the claim for examined states, and the claim for generated candidates naturally follows. We show that for each examined state S , $f(S) \leq s^*$. At any stage, if $f(TopK.root) = s^*$, $f(S) \leq s^*$ is ensured by the while condition on Line 3. If $f(TopK.root) > s^*$, the top- k results have not been completely retrieved. Since S appears at the root of g_heap and $f(S)$ is the lower bound value of all future tuples, we have $f(S) \leq s^*$. ■

While the number of examined states is type-I optimal, the number of generated candidates is significantly higher. Fixing the page size as $4kB$, the fanout of B -tree node is 204 [14]. The number of child states (i.e., Cartesian product) of two B -tree indices is up to 4.2×10^4 , and that of three B -tree indices is up to 8.5×10^6 . This is clearly non-trivial cost in terms of both CPU and memory. To avoid full expansion, an alternative method that uses unidirectional expansion was discussed in [15]. To expand a joined state $S = (a, b)$, either a is paired with all child nodes of b or b is paired with all child nodes of a . In this way, the number of child states is limited by the fanout of the node. However, the states generated by unidirectional expansion are less precise than those generated by full expansion, and thus the access scheduling may not be optimal. We identify the first challenges: *how to efficiently generate candidate states while preserving the optimal (type-I) access scheduling?*

3.3.2 Type-II Optimality

To achieve type-II optimality, we first characterize two types of states: *redundant state* and *empty state*. An index node may appear in many joint states, and thus may be requested for retrieval for multiple times. We say a leaf index node is redundant if it has been retrieved previously. A non-leaf index node is redundant if all child nodes are redundant. Consequently, a joint state is redundant if all composing index nodes are redundant. Redundant states need not to be retrieved because all data tuples contained by them have already been seen by the query processing algorithm. Since many index implementations buffer the previously retrieved index nodes, the redundant nodes (and thus the redundant states) can be identified on the fly. If the nodes are not buffered, the algorithm can maintain the IDs of those redundant index nodes in a hash-table.

To illustrate the definition of empty state, we build a mapping between data tuples and joint leaf states. We say a leaf state *contains* a tuple t if t appears in all the leaf index nodes joining the leaf state. A tuple may partially appear in many leaf states. However, there is a unique leaf state that contains it. On the other hand, a leaf state which contains one or more tuples is called *non-empty state*. Otherwise, it is an *empty state*. For example, in Figure 2.a, $(a1, b1)$ is an empty state and $(a2, b2)$ is a non-empty state. The definition of empty (non-empty) state can be recursively extended to non-leaf state as follows: A state is empty if all child states are empty; otherwise, it is non-empty. Let $S(t)$ be the leaf state containing tuple t . The following lemma shows a necessary condition of access scheduling using index-merge framework.

LEMMA 2. For any top- k query that is processed by the index-merge framework, suppose the final results are $t_1, t_2, \dots, t_k$, and $f(S(t_i)) < f(t_i)$ for all $i = 1, \dots, k$. The leaf states $S(t_1), S(t_2), \dots, S(t_k)$ must be retrieved when the query execution terminates³.

PROOF. In Algorithm 1, the query processing terminates when $\max_{i=1}^k f(t_i) \leq f(g_heap.root)$, for current top- k results $t_1, \dots, t_k$. Assume $S(t_i)$ has not retrieved. We have $f(g_heap.root) \leq f(S(t_i))$, which contradicts with $f(S(t_i)) < f(t_i) \leq \max_{i=1}^k f(t_i) \leq f(g_heap.root)$. ■

Since each t_i in the top- k results will be retrieved from non-empty state $S(t_i)$, it is safe to prune those empty-states. Thus, *the type-II optimal scenario is that an algorithm only retrieves states S such that: (1) $f(S) \leq s^*$, (2) S is not empty, and (3) S is not redundant.* We refer to n_{II}^* as the optimal number of states given by type-II optimality. Following our example in Figure 2.b, we may skip state $(a1, b1)$, and only retrieve state $(A.root, B.root)$ and $(a2, b2)$.

We demonstrate the importance of pruning empty states by examining the leaf-states only. Suppose the database has $1M$ tuples, and each B_+ -tree index contains at least 5,140 leaf nodes (with fanout 204). Merging two indices leads to 2.64×10^7 leaf-states in total. Among them, there are at most $1M$ non-empty states. Thus, the probability that a leaf-state is empty is more than 96%.

The sort-merge framework also has empty-state problem, where objects near top on one list may rank low in other lists. To avoid accessing the large portion of data which is in the middle of the sorted lists, random access is used to directly jump to the bottom of the lists and resolve missing values. In the following, we discuss why random access is not applicable in the index-merge framework. As presented in the last subsection, tuples can be classified into three categories: fully merged, partially merged and unseen. Let s_k be the upper bound score for current top- k fully merged tuples, s_p and s_u be the lower bound scores for partially merged and unseen tuples. The search terminates when $s_k \leq \min(s_p, s_u)$. In sort-merge, we have $s_p \leq s_u$ because of the monotonicity of the ranking function. By issuing random accesses on partially merged tuples, s_k may decrease and s_p may increase. As the result, the termination condition may be satisfied without continuing to sequentially scan lists. While in index-merge, $s_u = s_p = f(g_heap.root)$ because for each partially merged and unseen tuple t , $S(t)$

³ $f(S(t_i)) = f(t_i)$ is possible for some special case. Here we simplify the analysis by assuming $f(S(t_i)) < f(t_i)$.

has not been retrieved. Issuing random accesses on partially merged tuples may decrease s_k to s'_k . However, since for all partially merged t , $f(t) \geq f(S(t)) \geq f(g_heap.root)$, we have $s'_k \geq f(g_heap.root) = s_u$. We conclude that random access in index-merge does not lead to early termination.

Different from redundant states, which can be checked on the fly, the empty state can only be identified with the assistance of some pre-computed module. Moreover, this module needs to be light-weighted. We identify the second challenge: *how to effectively prune empty-states with low overhead?*

4. PROGRESSIVE MERGE

This section presents our solution for the first challenge. In Algorithm 1, to search for the best child state, a parent state is fully expanded (on Line 8). In fact, most of them are never examined (on Line 4). For example, in Figure 2.a, there are 9 child states expanded by $(A.root, B.root)$, and only $(a1, b1)$ and $(a2, b2)$ are examined. In fact, to ensure the type-I optimality, we only need to compute the *next best* child state, for a given state S . We abstract⁴ the interface that fulfills this requirement as $S.get_next$.

In the rest of this section, we first discuss a double-heap method that integrates $S.get_next$ with Algorithm 1, and then discuss two implementations for $S.get_next$.

4.1 The Double-heap Algorithm

Let $S.num$ indicate how many times the $S.get_next$ is called. In our example, suppose $S = (A.root, B.root)$. The first call of $S.get_next$ returns $(a1, b1)$ and the second call of $S.get_next$ returns $(a2, b2)$. $S.get_next$ returns nothing when all child states are returned. For simplicity, we denote $S.num$ as the num^{th} best child state of S according to $f(S.num)$.

Before we proceed to the implementation of $S.get_next$, we first discuss how to integrate $S.get_next$ to Algorithm 1. To distinguish the integrated algorithm from Algorithm 1, we refer to the new algorithm as *dheap* (i.e., double-heap). Instead of fully expanding S , and then discarding S (on Line 8 of Algorithm 1), *dheap* gets $S.num$ (by calling $S.get_next$) and inserts both $S.num$ and S to g_heap . We keep S in g_heap because the query execution may further request the next best child state of S . Consequently, to reflect the status of partial expansion, we update $f(S) = f(S_{num+1})$, which is the best possible score for all future child states. Following our example in Figure 2, when $S = (A.root, B.root)$ needs to be expanded, we get $S_1 = (a1, b1)$, update $f(S) = f(S_2) = f(a2, b2)$ and insert both S_1 and S into g_heap .

The complete framework for progressive and selective index merge is outlined in Algorithm 2, where Lines 1 to 15 are the updated query processing procedure, and Lines 21 to 34 are the $S.get_next$ method. Particularly, lines 5-6, 22-24 and 32-33 exploit pruning empty-states by join-signatures. The code is put here for the completeness of the algorithm, and the details will be discussed in the next section. The key components are *neighborhood.expand* and *threshold.expand*. The former works for some special scenario such that the best child state can be analytically found; and the latter is applicable on general cases. To avoid generating the same states along the multiple calls of $S.get_next$, each S maintains the status of child states that have been already generated, using a heap (i.e., l_heap in $S.get_next$).

⁴Here we use the object-oriented programming convention.

We refer to this heap as *local heap*, in contrast to the *global heap* (i.e., g_heap on Line 2) used in the main query processing loop. In the following subsections, we describe the two progressive search strategies one by one. For simplicity, we demonstrate both methods by merging two B -tree indices. The generalization to multiple indices is straightforward.

Algorithm 2 Progressive and Selective Merge

Input: A set of Indices I , ranking function f , top- k

```

1:  $TopK = \phi$ ; //heap with size  $k$  to hold current top- $k$ 
2:  $g\_heap = \{Joint\ Root\}$ ; //heap for state search
3: while ( $g\_heap \neq \phi$  and  $f(TopK.root) > f(g\_heap.root)$ )
4:   remove top entry  $S$  from  $g\_heap$ ;
5:   if ( $S$  is empty or redundant)
6:     continue;
7:   if ( $S$  is a leaf state)
8:     Retrieve data and update  $TopK$ ;
9:   else //  $S$  is a non-leaf joint state
10:     $next = S.get\_next()$ 
11:    if ( $next \neq null$ ) // non-empty non-redundant
12:      insert  $next$  to  $g\_heap$ ;
13:    if ( $S.l\_heap \neq \phi$ ) // more child states in  $S$ 
14:      insert  $S$  to  $g\_heap$ ;
15: return
```

Procedure $S.get_next()$

Vars: Local heap: l_heap

```

21: if ( $l\_heap = \phi$ ) // the first time  $S.get\_next$  is called
22:   load state-signature  $sig$ ;
23:   if ( $sig = null$ ) //no signature for empty-states
24:     return null; //return nothing, keep  $l\_heap = \phi$ 
25:   load index nodes of  $S$ ;
26: if ( $f$  is (semi-)monotone in  $S$ ); //neighborhood
27:    $next = neighborhood.expand()$ ;
28: else //threshold
29:    $next = threshold.expand()$ ; //the best child state
30: if ( $l\_heap \neq \phi$ ) //there are child states left
31:    $f(S) = f(l\_heap.root)$ ; //  $l\_heap$  was updated
32: if ( $next$  is empty or redundant)
33:    $next = null$ ;
34: return  $next$ ;
```

4.2 Neighborhood Expansion

Although an ad-hoc ranking function does not perform regularly over the entire region, it may have monotonicity or semi-monotonicity in some sub-regions. We say a function f is *semi-monotone* if $f(x_1, x_2, \dots, x_m) \leq f(x'_1, x'_2, \dots, x'_m)$ whenever $|x_i - o_i| \leq |x'_i - o_i|$, for every i . f achieves minimal value at $(o_1, o_2, \dots, o_m)$. For example, Figure 3 shows the plot of function $f = Ae^{-A^2 - B^2}$ over two attributes A and B . The function is neither monotone nor semi-monotone over the entire domain. However, in sub-region $(A = [-2, 0], B = [-1, 1])$, f is semi-monotone; and in sub-region $(A = [1, 2], B = [-2, -1])$, f is monotone. The index-merge method recursively partitions the joint space and it is very likely that an ad-hoc function f becomes monotone or semi-monotone within a joint state. We present the neighborhood expansion for these special cases.

Since the neighborhood expansion requires a total order to be defined on the entries of an index node, the method may

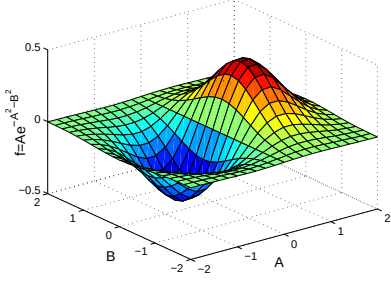


Figure 3: Local Monotonicity

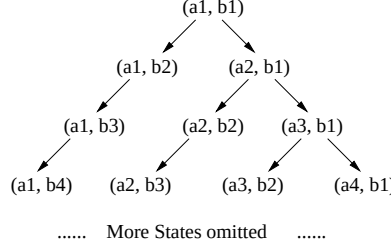


Figure 4: Neighborhood Expansion

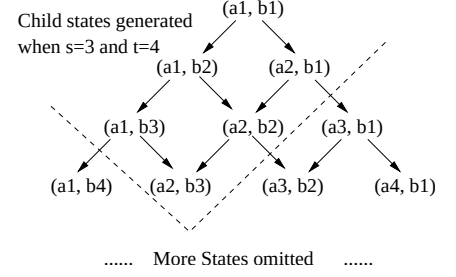


Figure 5: Threshold Expansion

not be used on R -tree indices. Let $S = (A_1, B_1)$ be the state to be expanded. $a_1, \dots, a_n$ and $b_1, \dots, b_l$ are child entries of index node A_1 and B_1 , respectively. Without loss of generality, we assume both a_i ($i = 1, \dots, n$) and b_i ($i = 1, \dots, l$) are sorted by attribute values. Suppose f is monotone over S and it achieves minimal value on (a_1, b_1) (i.e., initial state, referred as $I(S)$). Thus, we have $f(a_i, b_j) \leq \min(f(a_{i+1}, b_j), f(a_i, b_{j+1}))$. The neighborhood expansion starts with the initial state and progressively generates neighboring states. A straightforward definition for the neighboring states of (a_i, b_j) is to enclose both (a_{i+1}, b_j) and (a_i, b_{j+1}) . Since a state (a_{i+1}, b_{j+1}) can be generated by either (a_i, b_{j+1}) or (a_{i+1}, b_j) , this approach requires duplicate-checking, which incurs additional overhead. Alternatively, we can define the neighborhood of a child state (a_i, b_j) as:

$$N(a_i, b_j) = \begin{cases} \{(a_i, b_{j+1})\} & \text{if } 1 < j < l \\ \{(a_{i+1}, b_j), (a_i, b_{j+1})\} & \text{if } j = 1, i < n \\ \emptyset & \text{otherwise} \end{cases}$$

The definition of N is illustrated in Figure 4. Clearly, there is no duplicate states to be generated. Whenever (a_i, b_j) appears at the root of the local heap and is to be returned by $S.get_next$, we will insert $N(a_i, b_j)$ into the local heap. The procedure of neighborhood expansion is displayed as *neighborhood_expand* in Algorithm 3.

We briefly discuss how to extend the method to semi-monotone f . Suppose the extreme point of f is $o^* = (x^*, y^*)$. If o^* falls in a child state (a_s, b_t) , the initial state is $I(S) = \{(a_s, b_t)\}$. Otherwise, we can find an i (j) such that a_i and a_{i+1} (b_j and b_{j+1}) enclose x^* (y^*). Accordingly, $I(S) = \{(a_i, b_j), (a_{i+1}, b_j), (a_i, b_{j+1}), (a_{i+1}, b_{j+1})\}$. We define the neighborhood for $I(S) = \{(a_s, b_t)\}$ case as follows, the definition for the other case is similar:

$$N(a_i, b_j) = \begin{cases} \{(a_i, b_{j+1})\} & \text{if } t < j < l \\ \{(a_i, b_{j-1})\} & \text{if } 0 < j < t \\ \{(a_{i+1}, b_j), (a_i, b_{j+1}), (a_i, b_{j-1})\} & \text{if } j = t, s < i < n \\ \{(a_{i-1}, b_j), (a_i, b_{j+1}), (a_i, b_{j-1})\} & \text{if } j = t, 0 < i < s \\ \{(a_{i-1}, b_j), (a_i, b_{j+1}), (a_{i+1}, b_j), (a_i, b_{j-1})\} & \text{if } i = s, j = t \\ \emptyset & \text{otherwise} \end{cases}$$

In general, to merge m indices, the cardinality of neighborhood for monotone functions is up to m and that for semi-monotone functions is up to $2m$. Lemma 3 gives the computational performance of the neighborhood expansion.

LEMMA 3. Suppose f is monotone (or semi-monotone) over the entire domain, and there are m indices to be merged. The number of states generated by neighborhood expansion is upper bounded by mn_I^* for monotone functions and $2mn_I^*$ for semi-monotone functions, where n_I^* is the type-I optimal number of states to be generated (Section 3.3.1).

PROOF. According to Lemma 1, $S.get_next$ will be called at most n_I^* times in Algorithm 2. Each time when the $S.get_next$ is called, the neighborhood expansion will generate up to m child states for monotone f and up to $2m$ child states for semi-monotone f . The conclusion follows. ■

Algorithm 3 Neighborhood and Threshold Expansions

Procedure *neighborhood_expand*()

```
41: if ( $l\_heap = \emptyset$ ) // the first time  $S.get\_next$  is called
42:    $l\_heap = I(S)$ ; // insert initial states
43: remove top entry  $next$  from  $l\_heap$ ;
44: insert  $N(next)$  to  $l\_heap$ ;
45: return  $next$ ;
```

Procedure *threshold_expand*()

Vars: $S = (n_1, \dots, n_m)$, and $n_i = (e_i^1, \dots, e_i^{M_i})$ for each i
Current threshold position t_i ($i = 1, \dots, m$)

```
51: if ( $l\_heap = \emptyset$ ) // the first time  $S.get\_next$  is called
52:    $l\_heap = (e_1^1, \dots, e_m^1)$ ; // insert initial states
53:    $t_1 = \dots = t_m = 2$ ; //initial threshold positions
54:    $find\_next()$ ; //find the first candidate
55: remove top entry  $next$  from  $l\_heap$ ;
56:  $find\_next()$ ; //search for next state
57: return  $next$ 
```

Procedure *find_next*()

```
61: while ( $f(l\_heap.root) > \min(f'(e_1^{t_1}), \dots, f'(e_m^{t_m}))$ )
62:   and ( $\exists i \in \{1, \dots, m\}$  such that  $(t_i \leq M_i)$ )
63:    $s = \arg \min_{i=1}^m f'(e_i^{t_i})$ ;
64:    $news = [e_1^1, \dots, e_1^{t_1-1}] \times \dots \times [e_s^{t_s}] \dots [e_j^1, \dots, e_j^{t_m-1}]$ ;
65:    $t_s++$ ;
66:   for each  $cs$  in  $news$ 
67:     if ( $cs$  is not empty or redundant)
68:       insert  $cs$  to  $l\_heap$ ;
69: return;
```

4.3 Threshold Expansion

Here we discuss the more general threshold expansion. Different from the neighborhood expansion, where the initial and consequent child states can be analytically located, threshold expansion conducts searching over the child state space. Suppose the state to be expanded is (A_1, B_1) , and their entries are $a_1, a_2, \dots, a_n$ and $b_1, b_2, \dots, b_l$, respectively. We define $f'(a_i) = f(a_i, B_1)$ ($i = 1, \dots, n$), which is the best score that a_i can achieve by pairing with any b_j ($j = 1, \dots, l$). Similarly, we define $f'(b_j) = f(A_1, b_j)$ ($j = 1, \dots, l$). We sort a_i and b_j in ascending order of $f'(a_i)$ and $f'(b_j)$ values, respectively. Without loss of generality, we assume the sorted orders are $a_1, a_2, \dots, a_n$ and $b_1, b_2, \dots, b_l$.

The threshold expansion uses a sort-merge paradigm. We start search by inserting (a_1, b_1) to $lheap$ (i.e., local heap), and use two variables s and t to keep track of the next positions on a_i and b_j sorted lists (i.e., threshold position). Initially, $s = t = 2$. The stop condition is $f(lheap.root) \leq \min(f'(a_s), f'(b_t))$. At that time, the $lheap.root$ contains the next best child state. While the stop condition does not hold, the algorithm conducts progressive search. Suppose $f'(a_s) < f'(b_t)$, the algorithm creates $t - 1$ new child states (a_s, b_j) ($j = 1, \dots, t - 1$), inserts them to $lheap$, and then increases s by 1. For example, when $s = 3$ and $t = 4$, the states that have been generated are shown in Figure 5.

In general, suppose S is joined by m index nodes $(n_1, \dots, n_m)$, and each n_i consists of child entries $e_i^1, \dots, e_i^{M_i}$ (M_i is the fanout of node n_i). We define $f'(e_i^k) = f(n_1, \dots, e_i^k, \dots, n_m)$. The algorithm maintains a threshold position t_i for each n_i . Whenever a t_s is selected to advance, we generate the new child states by the Cartesian product $[e_1^1, \dots, e_1^{t_1-1}] \times \dots \times [e_s^1, \dots, e_s^{t_s-1}] \times \dots \times [e_m^1, \dots, e_m^{t_m-1}]$. The algorithm is shown in Algorithm 3 as *threshold_expand*.

A function f is *general* over the state S if for any t_i ($i = 1, \dots, m$), the lower bound of $f(e_1^{t_1}, \dots, e_m^{t_m})$ can only be derived by $\min_{i=1}^m (f'(e_i^{t_i}))$. Accordingly, we refer to algorithms performing child state search under this assumption as *general algorithms*. The following lemma shows that the threshold expansion is instance optimal [12] with optimal ratio 2^m , where m is the number of indices to be merged.

LEMMA 4. Assume f is general over state $S = (n_1, \dots, n_m)$, and for each node n_i , its entries satisfy $f'(e_i^1) < f'(e_i^2) < \dots < f'(e_i^{M_i})$. The *threshold_expand* (or *te*) is instance optimal such that $n_{te} \leq 2^m n_{alg}$, where n_{te} is the number of child states generated by *te*, n_{alg} is the number of child states generated by any other general algorithm *alg* that correctly finds the next best child state.

PROOF. Suppose *te* halts at t_i on each node n_i , and the next best child state is $nb = (e_1^{s_1}, \dots, e_m^{s_m})$. Clearly, $s_i < t_i$ for all $i = 1, \dots, m$. We show that for all i , $f'(e_i^{t_i-2}) < f'(e_i^{t_i-1}) \leq f(nb)$. Assume the threshold position at node n_k ($k \neq i$) is t'_k when *te* decides to advance to $t_i - 1$ on node n_i . If $f'(e_i^{t_i-1}) > f(nb)$, we have $s_i < t_i - 1$ and there must exist one k such that $s_k \geq t'_k$. Otherwise, the best state nb has already been generated by *te* and $t_i - 1$ will not be selected. On the other hand, we have $f'(e_i^{t_i-1}) \leq f'(e_k^{t'_k}) \leq f'(e_k^{s_k}) \leq f(nb)$, which leads to contradiction. Since f is *general* over S , any general algorithm *alg* that correctly finds the best state has to generate and check all states from $[e_1^1, \dots, e_1^{t_1-2}] \times \dots \times [e_m^1, \dots, e_m^{t_m-2}]$. Thus, $n_{te} = \prod_{i=1}^m (t_i - 1) \leq 2^m \prod_{i=1}^m (t_i - 2) = 2^m n_{alg}$. ■

5. SELECTIVE MERGE

In this section, we address the challenge towards the type-II optimality. To prune empty-states in merging indices $I_1, I_2, \dots, I_m$, we materialize a *join-signature* of these m indices. Suppose the database has J indices. An ideal scenario is to compute a join-signature on any combination of two or more indices, and this leads to $(2^J - J - 1)$ different join-signatures in total. In reality, one can only compute join-signatures on index combinations which are possibly to be queried. Alternatively, one can also materialize join-signatures on each pair of indices only, and use them to answer arbitrary queries. In the rest of this section, we discuss what is join-signature, how to compute join-signature and how to use join-signature during query processing.

5.1 Join-Signature

The join-signature is composed by the *state-signatures* of all non-leaf and non-empty states. For each non-leaf and non-empty state $S = (n_1, \dots, n_m)$, we define $card(S) = \prod_{i=1}^m M_i$ as the cardinality of the child states, where M_i is the fanout of node n_i . The state-signature is an m -way bit array, where each entry corresponds to a child state. If a child state is not empty, the entry is set to 1; otherwise, it is 0. Using our example in Figure 2, only $S = (A.root, B.root)$ is a non-leaf and non-empty state. The state-signature of S is a 3×3 bit array, as shown in Figure 6.

	b1	b2	b3
a1	0	1	1
a2	1	1	0
a3	1	0	1

<i>tid</i>	<i>A.path</i>	<i>B.path</i>
t1	$\langle 1 \rangle$	$\langle 2 \rangle$
t2	$\langle 1 \rangle$	$\langle 3 \rangle$
t3	$\langle 1 \rangle$	$\langle 3 \rangle$
t5	$\langle 2 \rangle$	$\langle 1 \rangle$
t4	$\langle 2 \rangle$	$\langle 2 \rangle$
t6	$\langle 3 \rangle$	$\langle 1 \rangle$
t7	$\langle 3 \rangle$	$\langle 1 \rangle$
t8	$\langle 3 \rangle$	$\langle 3 \rangle$

Figure 6: signature

Table 3: Pathes on Indices

The space consumption of the join-signature depends on the number of non-leaf and non-empty states, as well as the size of each state-signature. We discuss these two issues one-by-one. As defined in Section 3.3.2, a leaf-state is non-empty if it contains at least one tuple. On the other hand, since tuples are exclusively distributed into different leaf-nodes in an index, tuples are also exclusively contained by different leaf-states. Suppose the database has T tuples. We conclude that there are at most T non-empty leaf-states, and at most $(d - 1)T$ non-empty and non-leaf states (where d is the maximum depth among all joined indices).

We then discuss how to control the size of each state-signature, such that each of which can be stored within size P . Typically, P can be set as the page size. Given a state S , the value of $card(S)$ (i.e., the number of child states) varies significantly from merging two indices to merging three indices. If $card(S) \leq P$, we store the state-signature as it is, and possibly, put neighboring state-signatures together if their accumulative size does not exceed P . Since state-signatures are essentially bit-maps, one can further apply bit-map compression methods, such as run-length encoding [14] and prefix-compression encoding [13], to reduce the sizes. We omit the details in this paper.

When $card(S) > P$, we use bloom filter [3] for compres-

sion. Bloom filter uses k hash functions and maps an entry to k positions in a bit array. During the building phase, the bits at those k positions are set to 1. At the query time, one can apply the k same hash functions to get k positions, and return true if and only if all of them are 1. False positives are possible. But it guarantees no false negative. To use bloom filter for state-signature, we set the array size as $b(\leq P)$, and insert to the bloom filter all “1” entries in the state-signature. During query execution, if the bloom filter returns false for a child state c , c must be an empty-state. Suppose the number of non-empty child nodes (i.e., “1” entries) is ne . The optimal choice [3] for k is $\frac{b}{ne} \ln 2$. To control the computational complexity (i.e., the number of hash functions), we set the maximum number of hash function as k . Consequently, $b = \min(P, \frac{k \times ne}{\ln 2})$.

To reference a state-signature, we compute a unique key for each state $S = (n_1, n_2, \dots, n_m)$ as follows. A level- k node n is associated with a path from the root (level 1) to n ’s parent (level $k - 1$). The path consists of a sequence of entry positions: $path(n) = \langle p_1, p_2, \dots, p_{k-1} \rangle$, where p_i is the entry position at the level- i node. For example, node a_1 in index A (Figure 1) corresponds to $path(a_1) = \langle 1 \rangle$. The key of S is a combination of paths of n_i : $key(S) = (path(n_1), \dots, path(n_m))$. In implementation, one can one-to-one map $key(S)$ to a string or an integer. For each join-signature, we build an index on $key(S)$ for state-signatures.

5.2 Computing Join-Signatures

A naïve method to compute the join-signature is to traverse the joint state space and find the non-empty and non-leaf states. This is obviously not a scalable solution, since the space of joint states grows exponentially with the number of indices. Here we present a tuple-oriented approach.

Similar to the path generation for nodes, we can also compute a path for each tuple t . Given an index with depth d , the path of a tuple t is $path(t) = \langle p_1, \dots, p_d \rangle$. Since the joined space is defined on the node granularity, we only need to know which leaf-node contains t . Hence, we can ignore the position on the leaf node, and $path(t) = \langle p_1, \dots, p_{d-1} \rangle$. The paths for tuples in the sample database (Table 2) are shown in Table 3.

Treating each index as a dimension, and tuple paths as values, we compute the join-signature by recursive-sorting. Suppose we need to compute the join-signature for m indices $(I_1, \dots, I_m)$, and the depth of each index is d_i ($i = 1, \dots, m$). According to the path generation discussed above, each tuple path has $(d_i - 1)$ entries in dimension I_i . To compute the state-signature for joint root state, we sort the tuples according to the first path entry on each I_i (i.e., first compare $I_1.p_1$, then $I_2.p_1$, and so on). As an example, Table 3 is sorted by the first path entries. We then scan the sorted tuple list again, and insert distinct $(I_1.p_1, I_2.p_1, \dots, I_m.p_1)$ into the state-signature, which is implemented by a bit array or a bloom filter. For each sub-list of tuples that share the same $(I_1.p_1, I_2.p_1, \dots, I_m.p_1)$, we recursively sort it on the second path entry on I_i . At the same time, $(I_1.p_1, I_2.p_1, \dots, I_m.p_1)$ is the key to reference the next state-signature.

The above method is similar to some sorting-based data cube computation methods [2]. In fact, computing multiple join-signatures is indeed a multi-dimensional aggregation problem. As we mentioned earlier, suppose the database has J indices and we may compute up to $(2^J - J - 1)$ different join-signatures. Some techniques for efficient data cubing

can be directly applied in our problem. First, when database is too large to fit in memory, one can partition the database (e.g., by $I_1.p_1$) and compute each partial database [17]. Secondly, we can share the sorting on common indices [2] when computing multiple join-signatures. Finally, when J is large and there are too many join-signatures, we can only compute the low-dimensional (e.g., pairwise) join-signatures [21]. The low-dimensional join-signatures can partially fulfill the task to prune empty-states in high-dimensional index-merge, and this will be addressed in the next subsection.

5.3 Pruning Empty-State by Join-Signature

After presenting the join-signature and its computation, we now discuss how to use it to prune empty-states during query processing. Candidate states are generated by the $S.get_next$ procedure. To avoid generating empty-states, join-signatures can be integrated into $S.get_next$ procedure as shown in Algorithm 2. When $S.get_next$ is called for the first time, we load the state-signature for S using $key(S)$ (Line 22). For each candidate child state to be returned (i.e., $next$), we check with the state-signature on Line 33. A *null* state will be returned if $next$ is an empty state or a redundant state.

An even better solution is to push the empty-state checking into the *threshold.expand* and *neighborhood.expand*. For *threshold.expand*, we verify each child state before it is inserted into l_heap (Line 67, Algorithm 3). However, for *neighborhood.expand*, if a child state cs in $N(next)$ is empty, we still need to keep cs in the l_heap . This is because we may need to further expand to $N(cs)$, which may be non-empty, non-redundant and only accessible by cs .

A state-signature may be implemented by a bloom filter, which has false positives such that an empty state will be falsely recognized as a non-empty state. Suppose S is a non-leaf empty-state and was falsely passed the signature checking. At certain stage, S may be scheduled for expansion. Since $key(S)$ does not appear in join-signature, the algorithm notices that S is an empty state and the previous false positive will be corrected. For this purpose, we directly return *null* from $S.get_next$ (Line 24 of Algorithm 2). Since $S.l_heap$ is empty, S will also be discarded by the main query processing loop (Line 13 of Algorithm 2). In this way, the false positives on non-leaf states will not propagate. We have the following lemma with proof omitted.

LEMMA 5. In Algorithm 2, the expect number of states retrieved is $(n_I^* - n_{II}^*)fp + n_{II}^*$, where $n_I^* = |\{S | f(S) \leq s^*\}|$ is the type-I optimal state number (Section 3.3.1), $n_{II}^* = |\{S | f(S) \leq s^* \text{ and } S \text{ is neither empty nor redundant}\}|$ is the type-II optimal state number (Section 3.3.2), and fp is the expected false positive rate of the join-signature.

When the original bit-array is used (i.e., $fp = 0$), the number of states retrieved by Algorithm 2 is type-II optimal. Similarly, the expect number of disk accesses for state-signatures (referred as d_s) is $(n_I' - n_{II}')fp + n_{II}'$, where n_I' and n_{II}' are the number of non-leaf states with respect to type-I and type-II optimality, respectively. Since the majority states retrieved are leaf-states, we expect that d_s is far less than n_{II}' . On the other hand, retrieving a state S does not necessarily incur disk accesses because the index nodes involved in S are shared by other states and may have already been retrieved. Hence, the number of index node accesses (referred as d_i) may also be less than n_{II}' . In general, we

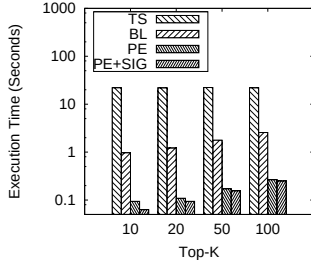


Figure 7: Execution Time w.r.t. K , $f = f_s$

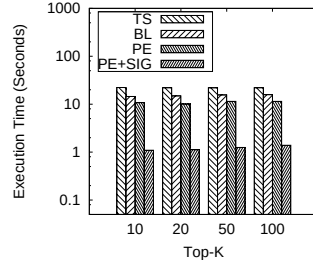


Figure 8: Execution Time w.r.t. K , $f = f_g$

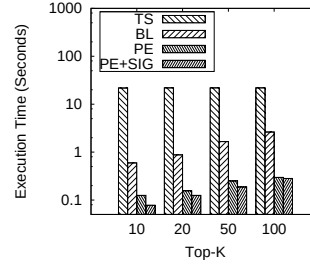


Figure 9: Execution Time w.r.t. K , $f = f_c$

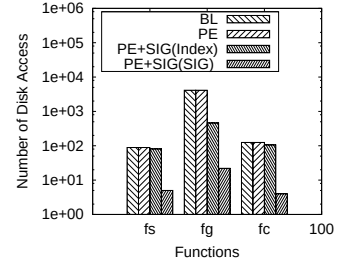


Figure 10: Disk Access w.r.t. f , $k = 100$

observe $d_s \ll d_i$. Moreover, without state-signature, d_i will be significantly higher (e.g., Table 1).

As mentioned in Section 5.2, one can use low-dimensional join-signatures to answer high-dimensional queries. Suppose the system pre-computed join-signatures on all pairs of indices. If a query involves $m > 2$ indices, for each state $S = (n_1, n_2, \dots, n_m)$, we will load state-signatures of states $S_{ab} = (n_a, n_b)$ (for all $a, b = 1, \dots, m$ and $a \neq b$). A child state is an empty state if any low-dimensional state-signature returns false. The low-dimensional join-signature may also speed-up child state generation in *threshold_expand* procedure in that whenever a pair of child entries (e_a, e_b) is identified as empty, all child states that are super-sets of (e_a, e_b) can be safely pruned (on Line 64 of Algorithm 3).

6. PERFORMANCE STUDY

This section reports our experimental results. We compare the query performance among four different methods: the *table scan* (*TS*) approach that sequentially scans the data file and computes top- k ; the *baseline* (*BL*) index-merge approach using Algorithm 1; the *progressive expansion* (*PE*) approach with the double heap algorithm only; and the *progressive expansion and join-signature* (*PE+SIG*) approach that applies both double heap algorithm and join-signatures. We first discuss the experimental setting.

6.1 Experimental Setting

We use both synthetic and real data sets for the experiments. The real data set is a variation of the *Forest Cover-Type* data set obtained from the UCI machine learning repository web-site (www.ics.uci.edu/~mllearn). This data set contains 1,162,024 data points with 6 selected attributes (cardinalities 255, 207, 185, 1,989, 5,787 and 5,827). We also generate a number of synthetic data sets for our experiments. The *TS* approach sequentially reads tuples from file. In the meanwhile, *TS* maintains a heap with size k to keep track the current top- k results seen so far. For other approaches using index-merge framework, we assume the attributes involved in ranking functions are indexed by either B_+ -trees or R -trees. By default, the page size in index nodes is set as 4KB. All methods are implemented in JAVA.

6.2 Experimental Results

We use *execution time* as evaluation metric and conduct experiments to evaluate the query performance with respect to different ranking functions, different type of indices and the number of indices for merging. Guided by the query performance, we further examine how to configure indices

for efficient online query processing. Finally, we show the scalability of the proposed methods, including both online query and offline computation costs.

6.2.1 Query Performance w.r.t. Ranking Functions

Since the index-merge paradigm is motivated by supporting non-monotonic ranking functions, we first evaluate query performance with respect to different types of functions. Suppose the ranking function is formulated on two attributes A and B , and each of them is indexed by a B_+ -tree. For demonstration, we use three queries with controlled functions: (1) a *semi-monotone* query with function $f_s = (A - a)^2 + (B - b)^2$ where a and b are random parameters. This is a typical nearest neighbor query, which is frequently used in database systems; (2) a *general* query with function $f_g = (A - B^2)^2$. This query is often used to measure the min square error; (3) a *constrained* query with function $f_c = \frac{A+B}{\eta(B)}$, where $\eta(B) = 1$ if $b_1 \leq B \leq b_2$, and $\eta(B) = 0$ otherwise. f_c essentially constraints the value of B to be between b_1 and b_2 , and b_1 and b_2 are two random parameters.

We use synthetic data sets in this set of experiments. By default, all data contains 1M tuples. The query execution time with respect to different k values and different ranking functions are shown in Figures 7 to 9. We observe that all the approaches using the index-merge framework perform better than table scan, while the speed-up margin differs from each ranking function: with f_s and f_c , both *PE* and *PE+SIG* are almost one order of magnitude faster than *BL*, which is already one order of magnitude faster than *TS*; and with f_g , *PE+SIG* is 10 times faster than *BL* and *PE*, which are only around 2 times faster than *TS*. The experimental results show the effect of two optimization techniques (progressive merge and selective merge) with respect to different ranking functions.

To explain the difference, we first analyze the properties of the three index-merge based approaches. The overall execution time can be decomposed into two parts: CPU time for state search and state generation; and I/O time for node retrieval. *PE* improves *BL* with respect to CPU cost, and *PE+SIG* further improves *PE* in terms of I/O cost. We then examine the three ranking functions to see which cost (i.e., CPU or I/O) dominates the overall performance. In f_s , the top answers are close to point (a, b) , and it is very likely that only a few index nodes need to be retrieved. Thus the I/O may be relatively cheap and the CPU cost dominates. The same observation is also applied on f_c . The hypothesis is well supported by Figures 7 and 9 in that: (1) *BL* is significantly faster than *TS* because *BL* requests much less

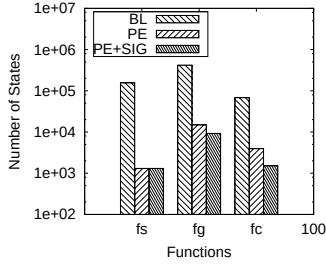


Figure 11: States Generated w.r.t. f , $k = 100$

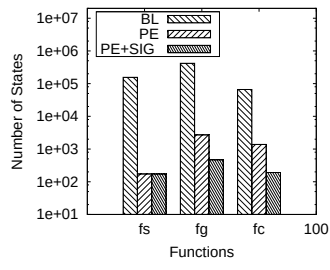


Figure 12: Peak Heap Size w.r.t. f , $k = 100$

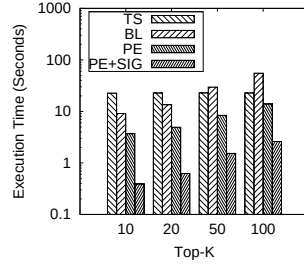


Figure 13: Execution Time w.r.t. K , Real Data

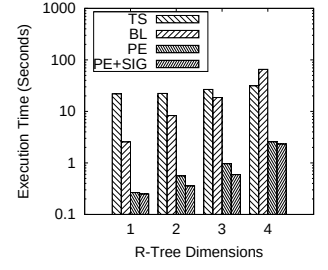


Figure 14: Execution Time w.r.t. R-Tree

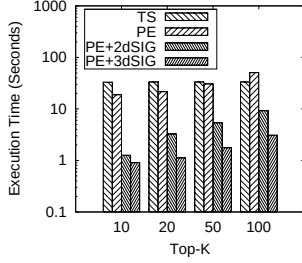


Figure 15: Execution Time w.r.t. K , 3 Indices

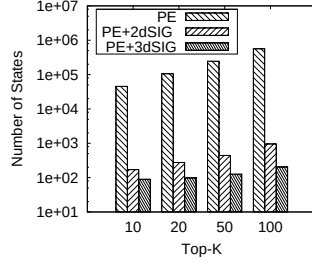


Figure 16: Peak Heap Size w.r.t. K , 3 Indices

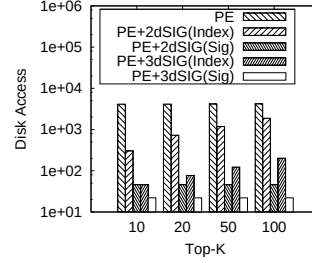


Figure 17: Disk Access w.r.t. K , 3 Indices

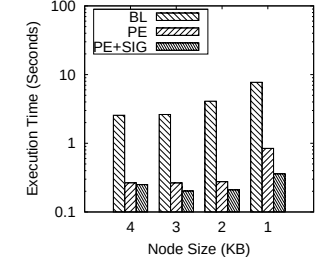


Figure 18: Execution Time w.r.t. Node Size

I/O; (2) PE further improves BL by order of magnitude because CPU cost dominates in BL ; and (3) the speed-up of $PE+SIG$ over PE is limited because the room for improvement is small (i.e., I/O is already very cheap). On the other hand, f_g is a difficult query function since the top results could scatter over the whole domain, and one has to retrieve more data to answer the query. As the result, the I/O cost dominates. As shown in Figure 8, the speed-up of both BL and PE over TS is not significant. $PE+SIG$ achieves substantial improvement because the join-signature prunes many empty-states, and thus reduces the I/O requests.

Figure 10 shows the number of disk access for three functions when $k = 100$. For $PE+SIG$, we further plot the number of index node requests and that of the state-signature requests. Comparing with the I/O for index nodes, the I/O cost for join-signatures is much less. Among the three functions, f_g incurs most I/O costs, and this is consistent with the above analysis. Figure 11 shows the number of generated states. We observe that the progressive expansion is quite effective in that it generates much less states. Sometimes $PE+SIG$ generates less states because the pruning of empty states (and whose child states). Finally, Figure 12 shows the peak heap size. The heap size of PE and $PE+SIG$ are computed by the accumulative size of the global heap and all local heaps. Note that even for f_g , the peak heap size of PE ($PE+SIG$) is 2,714 (469) when $k = 100$. This is actually a very important property to support in-memory computation. We will further address this in Section 6.2.3.

6.2.2 Query Performance on R-Tree Indices

After reporting the results on B_+ -tree indices, we evaluate the query performance on R -tree indices in this subsection. We have demonstrated the differences of three ranking functions in the last subsection and their behaviors are similar with R -tree indices. For simplicity, we only use f_s in

the following experiments. Suppose an R -tree index consists of d dimensions. Merging two R -tree indices means there are $2d$ attributes in the ranking function. We define $f_s = \sum_{i=1}^{2d} (A_i - a_i)^2$, where A_i is an attribute value and a_i is the query parameter. It is possible that some attributes are not involved in ranking, and we will address this in Section 6.2.4. As we discussed in Section 4, neighborhood expansion is not applicable in R -tree, since the nodes are not fully ordered. We use threshold expansion only.

We first conduct experiments on the real data set, whose 6 attributes are evenly divided into 2 groups. Each group is indexed by an R -tree. We vary the value of k from 10 to 100, and the query execution time is shown in Figure 13. Clearly, $PE+SIG$ performs best among all approaches. An interesting observation is the BL is even worse than TS when $k \geq 50$. This is because the ranking function involves 6 attributes, which make it more difficult to search for the final results. To verify this, we generate 4 different data sets with 2, 4, 6 and 8 dimensions, and build 2 R -tree indices (by evenly partition the attributes) on each of them. The execution time for $k = 100$ is shown Figure 14. As expected, it is more expensive to answer queries with more attributes. However, even for 4d R -tree, PE (and also $PE+SIG$) finishes query in around 2.5 seconds, which is more than 12 times faster than that by TS (31.5 seconds).

6.2.3 Query Performance on 3-Way Merge

All the previous experiments are conducted upon 2-way index merge. Here we examine query performance on 3-way index merge. We use $f_s = \sum_{i=1}^3 (A_i - a_i)^2$, where A_i are three attributes, and each of which is indexed by a B_+ -tree. As discussed in Section 5.3, the $PE+SIG$ approach has two choices: (1) use one 3d join-signature; or (2) use three 2d join-signatures (e.g., (A_1, A_2) , (A_1, A_3) and (A_2, A_3)). We report query performance for both scenarios.

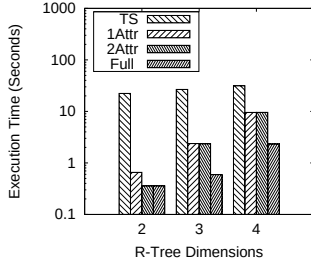


Figure 19: Partial Attributes in Ranking

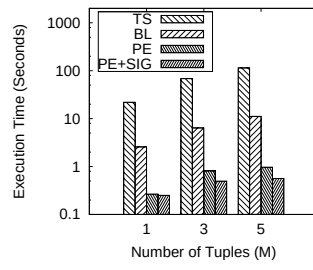


Figure 20: Execution Time w.r.t. T

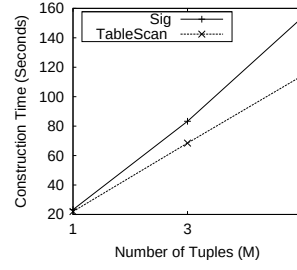


Figure 21: Construction Time w.r.t. T

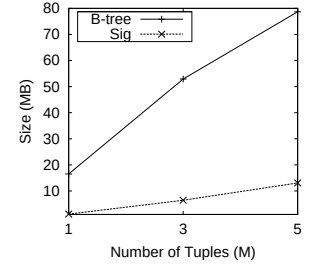


Figure 22: Size of Join-signatures w.r.t. T

Figure 15 shows the execution time on a synthetic data set with 3 dimensions. We did not report results of *BL*, because *BL* generates too many states and runs out of memory. Although *PE* can effectively control the heap size, its execution time becomes worse than *TS* when $k = 100$. Both *PE+SIG* approaches run significantly faster. Particularly, the *PE+SIG* with three *2d* join-signatures, although not as effective as that with one *3d* join-signature, performs very well in pruning empty states.

Setting $k = 100$, we plot the peak heap sizes in Figure 16 and the number of disk access in Figure 17. Clearly, the margin between *PE* and *PE+SIG* becomes much larger comparing with that in 2-way index merge (Figures 12 and 10). This is because the space of joint state grows exponentially with the number of merging indices. Consequently, both CPU cost for state search and I/O cost for index node retrieval are higher. Moreover, given the large number of candidate states, the probability that a state is not empty drops exponentially. *PE+SIG* achieves significant gain by pruning empty states. We also observe that in both *PE+SIG* approaches, the number of join-signature requests is several times less than that of index node requests.

6.2.4 Index Configuration

Having observed that merging multiple indices introduces high computational complexity in the above subsection, here we discuss how to alleviate the challenges with proper index configuration. Suppose the database consists m dimensions, and the query attributes are randomly selected. One can build m B_+ -trees on each attribute. Alternatively, one can group attributes with size r and build $\frac{m}{r}$ R -trees on each group. For example, when $m = 8$, we can set $r = 4$ and build two R -tree indices.

In Section 6.2.2, we have shown that the ranking functions consisting 8 attributes can be answered fairly efficiently by merging R -tree indices (Figure 14). The experiments are conducted under the assumption that all attributes in R -trees are involved in ranking function. In contrast to using low dimensional indices to answer high-dimensional queries, it is interesting to further check whether the high-dimensional indices (e.g., R -tree) can efficiently process low-dimensional queries. We construct three sets of f_s by selecting *one*, *two* and *all* attributes from each R -tree index, and run queries on the same data sets in Figure 14. The experimental results of *PE+SIG* (with $k = 100$) are shown in Figure 19. Not surprisingly, answering queries with full attributes is the most efficient. However, comparing with *TS*, the performance of partial attributes is still very attractive. For example, when $m = 6$ and $r = 3$, the execution time is

around 2.4 seconds for one or two attributes. For the same query, *TS* needs more than 26 seconds.

For some database with moderate number of ranking attributes (e.g., 4-8), our experiment results suggest that partitioning attributes into two groups and building R -tree index on each of them provides fairly robust query performance for queries involving any subset of attributes.

As part of the index configuration, we also test the query performance by varying the index node size. Typically, the node size is chosen from 1KB to 4KB. We generate a *2d* synthetic data sets, and build B_+ -tree indices with size 1KB to 4KB on each attribute. The execution time for $k = 100$ is shown in Figure 18. With smaller node size, the number of nodes increases, and so does the number of empty-states. On the other hand, with larger node size, the fanout of each node increases, and so does the number of states to be enumerated. In general, *PE+SIG* considers both effects, and thus is not very sensitive to node size.

6.2.5 Scalability

The final set of experiments is to study the scalability of the proposed methods. We use synthetic data set with 2 B_+ -tree indices, and vary the number of tuples from 1M to 5M. The query execution time (with $k = 100$) in Figure 20 shows that all methods scale quite well. Besides the online query performance, we also report the construction and space costs for the join-signature in Figures 21 and 22. To compare with, we plot the query execution time used by *TS* in Figure 21, which shows that the join-signature can be computed fairly efficiently in that it is comparable to table scan. We also compare the size of the join-signature with the size of one B_+ -tree index in Figure 22, and observe that the size of join-signature is at least 6 times smaller.

7. DISCUSSION

We discuss two extensions of the proposed methods: (1) merging indices from multiple relations, and (2) using the index-merge framework to answer other preference queries.

7.1 Merge Indices from Multiple Relations

The methods developed in this paper can also be used to merge indices from multiple relations. Particularly, we discuss how to extend the method to join primary keys and foreign keys. Assume primary keys and foreign keys are stored together with attribute values in the leaf index nodes. Thus the join condition can be evaluated during index merge.

While the double heap algorithm with progressive expansions can be directly used, there is a small variation to com-

pute the join-signatures. To construct the join-signature of index A from relation R_1 and index B from relation R_2 , we can first compute paths with respect to both indices for each tuple. Suppose each tuple in R_1 is associated with a path $path_A$ and each tuple in R_2 is associated with a path $path_B$. We then conduct a sort-merge join on R_1 and R_2 , and keep both $path_A$ and $path_B$ in the join results in R_3 . The method presented in Section 5.2 can be applied on R_3 to compute the join-signature.

7.2 General Preference Queries

Top- k queries are related to several other preference queries, such as skyline query [5] and convex hulls [4]. Skyline query asks for the objects that are not dominated by any other object in all dimensions. A convex hull query searches a set of points that forms a convex hull of all the other data objects. The methodology developed in this paper is also applicable to these queries. The key observation is that all the queries can be processed progressively in the top-down fashion. For demonstration, we discuss how to apply our method for skyline computation as follows. The method for convex hull queries is similar.

In [18], Papadias *et al.* developed a branch-and-bound search algorithm that progressively retrieves R -tree nodes, from root to leaves, until all the skylines are found. At any stage, if a node n is dominated by a data object, all the child nodes of n can be pruned. The same rule can be applied on the state space in this paper: If a state S is dominated by a data object, all the child states of S can be pruned. As soon as all the states in the global heap are pruned, the search for the skyline objects halts. The join-signature can be used without any modification, since the empty-states do not contribute to the final results either.

The progressive expansion methods are also applicable in skyline computation. Let $\bar{n}$ and $\underline{n}$ be the maximal and minimal attribute values among the region covered by n . Given a state $S = (A_1, B_1)$, with child nodes $(a_1, a_2, \dots, a_n)$ and $(b_1, b_2, \dots, b_m)$, we sort a_i and b_i according to $\underline{a_i}$ and $\underline{b_i}$, respectively. Similar to the threshold expansion in Section 4.3, we can progressively generate child states until a threshold position (r, t) , such that there exists a generated child state $c = (a^*, b^*)$ satisfying $a^* \leq \underline{a_r}$ and $b^* \leq \underline{b_t}$. Consequently, the $S.get_next$ method returns all the generated child states (instead of one child state in top- k query) and updates S 's coordinate values as $\underline{a_r}$ and $\underline{b_t}$.

8. CONCLUSIONS

For efficient query processing, we address two challenges within the index-merge paradigm. First, to reduce the search complexity, we develop a double heap algorithm that consists of the neighborhood expansion and the threshold expansion. Second, to avoid retrieving empty-state, we propose join-signature which is compact, easy to compute and incurs low overhead in query processing. Our performance evaluation shows that the proposed solutions improves the baseline solution by one order of magnitude.

There are many interesting research issues on further extensions of the index-merge methodology. For example, it will be useful to build a unified cost model for both state search and disk access so that the query processing can achieve the overall best performance. Another interesting problem is to further improve the index configuration by leveraging the workload information.

9. REFERENCES

- [1] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.
- [2] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD Conference*, pages 359–370, 1999.
- [3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [4] C. Bohm and H.-P. Kriegel. Determining the convex hull in large multidimensional databases. In *DaWaK*, pages 294–306. Springer-Verlag, 2001.
- [5] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
- [6] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD Conference*, pages 237–246, 1993.
- [7] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.
- [8] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects by exploiting relationships: computing top-k over aggregation. In *SIGMOD Conference*, pages 371–382, 2006.
- [9] S. Churdhuri and U. Dayal. An overview of data warehousing and data cube. *SIGMOD Record*, 26:65–74, 1997.
- [10] R. Fagin. Fuzzy queries in multimedia database systems. In *PODS*, pages 1–10, 1998.
- [11] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.
- [12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
- [13] A. Fraenkel and S. Klein. Novel compression of sparse bit-strings - preliminary report. *Combinatorial Algorithms on Words, NATO ASI Series*, 12:169–183, 1985.
- [14] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.
- [15] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD Conference*, pages 237–248, 1998.
- [16] S. Michel, P. Triantafyllou, and G. Weikum. Klee: a framework for distributed top-k query algorithms. In *VLDB*, pages 637–648, 2005.
- [17] K. Morfonios and Y. Ioannidis. Cure for cubes: cubing using a rolap engine. In *VLDB*, pages 379–390, 2006.
- [18] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.
- [19] H. Shin, B. Moon, and S. Lee. Adaptive and incremental processing for distance join queries. *IEEE Trans. Knowl. Data Eng.*, 15(6):1561–1578, 2003.
- [20] P. Valduriez. Join indices. *ACM Trans. Database Systems*, 12:218–246, 1987.
- [21] D. Xin, J. Han, H. Cheng, and X. Li. Answering top-k queries with multi-dimensional selections: The ranking cube approach. In *VLDB*, pages 463–475, 2006.
- [22] Z. Zhang, S. won Hwang, K. C.-C. Chang, M. Wang, C. A. Lang, and Y.-C. Chang. Boolean + ranking: querying a database by k-constrained optimization. In *SIGMOD Conference*, pages 359–370, 2006.
- [23] M. Zhu, D. Papadias, J. Zhang, and D. L. Lee. Top-k spatial joins. *IEEE Trans. Knowl. Data Eng.*, 17(4):567–579, 2005.