

P-Cube: Answering Preference Queries in Multi-Dimensional Space

Dong Xin¹, Jiawei Han²

¹Microsoft Research
Redmond, WA, U.S.A.

dongxin@microsoft.com

²University of Illinois at Urbana-Champaign
Urbana, IL, U.S.A.

hanj@cs.uiuc.edu

Abstract—Many new applications that involve decision making need online (*i.e.*, OLAP-styled) preference analysis with multi-dimensional boolean selections. Typical preference queries includes top- k queries and skyline queries. An analytical query often comes with a set of boolean predicates that constrain a target subset of data, which, may also vary incrementally by drilling/rolling operators. To efficiently support preference queries with multiple boolean predicates, neither *boolean-then-preference* nor *preference-then-boolean* approach is satisfactory.

To integrate boolean pruning and preference pruning in a unified framework, we propose *signature*, a new materialization measure for multi-dimensional group-bys. Based on this, we propose *P-Cube* (*i.e.*, data cube for preference queries) and study its complete life cycle, including signature generation, compression, decomposition, incremental maintenance and usage for efficient on-line analytical query processing. We present a signature-based progressive algorithm that is able to simultaneously push boolean and preference constraints deep into the database search. Our performance study shows that the proposed method achieves at least one order of magnitude speed-up over existing approaches.

I. INTRODUCTION

In many business applications, preference analysis (*e.g.*, top- k , skylines) is important for decision making. Moreover, it is often desirable to conduct analytical queries on different subsets of data. Given a relation R with two sets of dimensions: *boolean dimensions* and *preference dimensions*, a typical analytical query may specify a set of boolean predicates (on the set of boolean dimensions) that constrain a target subset of data, and a preference criterion (on the set of preference dimensions) that conducts the preference analysis.

The term of *preference query* was first proposed in [1], where the author presented a simple, logical framework for formulating preferences as preference formulas. In many database applications, preference queries have been studied in the context of skyline query, top- k query, convex hull query, and so on [2]. In this paper, we focus on two typical preference queries: top- k queries and skyline queries. However, the developed methodology applies to other types of preference queries as well. Given a set of n objects p_1, p_2, \dots, p_n , a *top- k query* returns k objects p'_i ($i = 1, \dots, k$) such that for any other objects p_j , $f(p_j) \geq \max_{i=1, \dots, k} f(p'_i)$, where f is a ranking function (assuming users prefer minimal values);

a *skyline query* returns all the objects p_i such that p_i is not *dominated* by any other object p_j , where dominance is defined as follows. Let the value of p_i on dimension d be $v(p_i, d)$. We say p_i is *dominated by* p_j if and only if for each preference dimension d , $v(p_j, d) \leq v(p_i, d)$, and there is at least one d where the equality does not hold. In the rest, we address the problem of *efficient processing preference queries with multi-dimensional boolean predicates*. Typical application scenarios include supporting top- k query in any subset of data (Ex. 1), or comparing skylines by drilling in multi-dimensional space to get deep insights of the data (Ex. 2).

Example 1: (Multi-dimensional top- k query) Consider a used car database (*e.g.*, kbb.com) with schema (*type, maker, color, price, mileage*). The first three are boolean dimensions, whereas the last two are preference ones. A user who wants to buy a used car with *type* = “sedan” and *color* = “red”, and also has expected *price* as $15k$ and expected *mileage* as $30k$ may issue the following top- k query:

select top 10 used cars from R

where type = “sedan” and color = “red”

order by $(price - 15k)^2 + \alpha(mileage - 30k)^2$

Here α is a user-specified parameter to balance the two options. ■

Example 2: (Multi-dimensional skyline comparison) Consider a digital camera comparison database (*e.g.*, bizrate.com) with schema (*brand, type, price, resolution, optical zoom*). Suppose the last three dimensions are preference dimensions. A market analyzer who is interested in canon professional cameras may first issue a skyline query with boolean predicate *type* = “professional” and *brand* = “canon”. The analyzer then rolls up on the *brand* dimension and checks the skylines of professional cameras by all makers. By comparing two sets of skylines, the analyzer will find out the position of canon cameras in the professional market. ■

There has been fruitful research work on efficiently processing top- k queries [3], [4], [5], [6] and skyline queries [2], [7], [8], [9], [10] in database systems. However, all these studies are performed under the context that the data consists of preference dimensions only. On the other hand, current

database management systems execute preference queries with boolean predicates by first retrieving data objects according to boolean selection conditions, and then conducting the preference analysis (e.g., top- k , or skyline). This approach is not efficient, especially when the database is large and the number of output is small. In conclusion, existing solutions conduct either boolean-only or preference-only search, and thus are not satisfactory.

In this paper, we propose a signature-based approach that *simultaneously pushes boolean and preference pruning in query processing*. More specifically, to address multi-dimensional boolean selections, we adopt the data cube model [11]. To support ad hoc preference analysis, we partition the data according to measure attributes. To seamlessly integrate multi-dimensional selection and preference analysis into a single framework, we develop a new measure, called *signature*, to summarize data partition for each multi-dimensional selection condition (e.g., a cell in the data cube). We refer to this signature-based materialization model as *P-Cube* (i.e., data cube for preference queries). *P-Cube* does not provide query answers directly. Instead, it is used to aid efficient query processing. The method is demonstrated on both skyline queries and top- k queries.

The rest of the paper is organized as follows. Section II discusses related work. Section III formalizes the problem. The *P-Cube* model is proposed in Section IV, and the query processing algorithms are developed in Section V. We report the experimental results in Section VI, and conclude the study in Section VII.

II. RELATED WORK

There have been extensive studies on efficiently computing skylines in database systems, including divide-and-conquer and block nested loop algorithms by Borzsonyi *et al.* [2], sort-first skyline by Chomicki *et al.* [7], bitmap and index methods by Tan *et al.* [10], nearest neighbor approach by Kossmann *et al.* [8] and branch-and-bound search by Papadias *et al.* [9]. All these approaches assume the queries consist of preference dimensions only, and the problem of skyline queries with boolean predicates was not well addressed. Similarly, although top- k query processing has been studied in both the middleware scenario [4], [3] and in the relational database setting [5], [6], these studies mainly discuss the configurations where only ranking dimensions are involved. The problem of top- k queries with multi-dimensional selections is not well addressed.

Our work is closely related to the recent work on ranking cube by Xin *et al.* [12]. In this work, each preference dimension is partitioned in equi-depth bins, and data tuples are grouped into grid cells by intersecting the bins from all participating preference dimensions. For each group-by, the cube stores $\langle cell_id, tuple_id_list \rangle$ pairs. During top- k query execution, the cell which contains the extreme point is first located, and the neighboring cells are progressively expanded. This search method is confined to grid partition and convex functions. The progressive expansion requires the neighbor

cells to be well defined, which is not the case in some other data partition methods (e.g., R-tree). Moreover, [12] did not discuss the incremental maintenance of the data cube.

Integrating boolean predicates with ranked queries was recently studied by Zhang *et al.* [13], which searches for the results by merging multiple indices. In their work, the boolean predicates are treated as part of the ranking function. An improved version of index merging was studied by Xin *et al.* [14], where the authors proposed progressive and selective merging strategy to reduce both CPU and I/O costs. In this paper, we will compare our approach to [14].

III. PROBLEM STATEMENT

Consider a relation R with *boolean dimensions* A_1, A_2, \dots, A_b , and *preference dimensions* N_1, N_2, \dots, N_p . The two sets of dimensions are not necessarily exclusive. A query specifies the boolean predicates on a subset of boolean dimensions and preference criteria on a subset of preference dimensions. The possible SQL-like notations for expressing top- k queries and skyline queries are as follows:

select top- k from R
where $A'_1 = a_1$ and \dots and $A'_i = a_i$
order by $f(N'_1, N'_2, \dots, N'_j)$

select skylines from R
where $A'_1 = a_1$ and \dots and $A'_i = a_i$
preference by N'_1, N'_2, \dots, N'_j

where $\{A'_1, A'_2, \dots, A'_i\} \subseteq \{A_1, A_2, \dots, A_b\}$ and $N'_1, N'_2, \dots, N'_j \subseteq \{N_1, N_2, \dots, N_p\}$.

Without losing generality, we assume that users prefer *minimal* values. The query results are a set of objects that belong to the data set satisfying the boolean predicates, and are also ranked high (for top- k) or not dominated by any other objects (for skylines) in the same set. For top- k queries, we assume that the ranking function f has the following property: *Given a function $f(N'_1, N'_2, \dots, N'_j)$ and the domain region Ω on its variables, the lower bound of f over Ω can be derived.* For many continuous functions, this can be achieved by computing the derivatives of f .

IV. P-CUBE

We first compare three different approaches, and motivate our solution: *P-Cube*. We then discuss a signature-based implementation.

A. Integrating Boolean and Preference Search

We focus on queries with hard multi-dimensional selections and soft user preference. To address the boolean selections, we adopt the data cube model, which has been popularly used for fast and efficient on-line analytical query processing (OLAP) in multi-dimensional space. A data cube consists of multi-dimensional cuboids (e.g., cuboids (*type*), (*color*), and (*type, color*) in the car database in Example 1), and each of which contains many cells (e.g., *type = sedan*). In traditional data cube, the query results for each cell are pre-computed. Our first proposal starts with the traditional data cube model,

and pre-computes all top- k results (or skylines) for each cube cell (e.g., *type = sedan*). Unfortunately, materializing all possible user preferred results is *unrealistic* since the user preference (e.g., ranking functions in top- k queries) is dynamic and not known until the query time.

In order to handle dynamic user preference, we discuss the second approach as follows. Instead of materializing the final query results, one may pre-compute data partition on preference dimensions (e.g., R -tree or grid partition on *price* and *mileage* in Example 1) for each cell. Given an ad-hoc ranking function, the query processing model can push the preference search by only visiting those promising regions (e.g., *price* around 15k and *mileage* around 30k in Example 1) in the partition. Unfortunately, this approach is *not scalable* because partitioning data is an expensive task and it will be conducted for all cells.

Instead of creating an individual data partitions for each cell, our third proposal only creates one data partition P over the preference dimensions (e.g., *price* and *mileage*). Using P as the template, one can *summarize* the data distribution over P for each cell by indicating which regions in the partition contain data belonging to the cell. For instance, P may partition data into n regions r_1, r_2, \dots, r_n . A cell (e.g., *type = sedan*) corresponds to a subset of data, which may only appear in $m \leq n$ regions r'_1, \dots, r'_m . A data summarization remembers those m regions. Comparing with a completely new data partition, the data summarization is much cheaper in both computational and storage cost. The data summarization is a bridge to join the boolean selection condition and preference search during online query processing. More specifically, the query algorithm progressively visits (in P) a sequence of regions that are promising for user preference (e.g., r_1^q, \dots, r_j^q), but skips those regions r_i^q which do not satisfy boolean selections (i.e., $r_i^q \notin \{r'_1, \dots, r'_m\}$). By doing this, the query processing pushes both boolean and preference pruning deep into the search. We call the data cube that uses data summarization as measure as P -Cube. In the rest of this section, we present *signature* as such kind of data summarization.

B. Signature-based Implementation

In this subsection, we discuss how to compute, retrieve and incrementally update signature.

1) *Signature Generation*: The computation of signature consists of three steps: *partition, summarization, compression and decomposition*. We demonstrate our methods using a sample database (Table I), where A and B are boolean dimensions, X and Y are preference dimensions, and *path* is computed to facilitate the signature computation.

Partitioning Data as Template: Data is partitioned according to the preference dimensions. We use R -tree [15] as an example. The same concept can be applied with other multi-dimensional partition methods. The details of R -tree partition can be found in [15], [16], and the R -tree partition of the sample database is shown in Figure 1.

Summarizing Data for Group-bys: Given the partition scheme, we generate a signature for each cell (e.g., $A = a_1$)

tid	A	B	X	Y	path
t1	a1	b1	0.00	0.40	(1, 1, 1)
t2	a2	b2	0.20	0.60	(1, 1, 2)
t3	a1	b1	0.30	0.70	(1, 2, 1)
t4	a3	b3	0.50	0.40	(1, 2, 2)
t5	a4	b1	0.60	0.00	(2, 1, 1)
t6	a2	b3	0.72	0.30	(2, 1, 2)
t7	a4	b2	0.72	0.36	(2, 2, 1)
t8	a3	b3	0.85	0.62	(2, 2, 2)

TABLE I
A SAMPLE DATABASE R

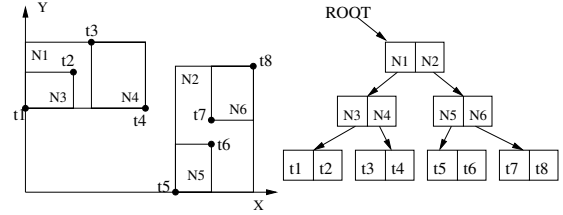


Fig. 1. R -tree Partition, $m = 1, M = 2$

as follows. We use one bit (i.e., 0/1) to indicate whether a data partition contains tuples belonging to a specified cell. For each node in the R -tree, we encode a bit array, where each bit corresponds to a child node. If there is no data belonging to the cell (e.g., $A = a_1$) in a child node, we set the bit value as 0 (1 otherwise). Figure 2.a shows an example signature for the cell ($A = a_1$), based on the partition scheme shown in Figure 1.

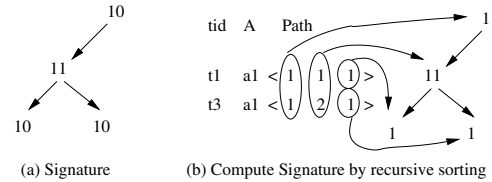


Fig. 2. ($A = a_1$)-signature

A data cube may consist of many cells, we can compute all signatures in a tuple-oriented way. Note that every tuple is associated with a unique path from the root. The path consists of a sequence of pointer positions: $\langle p_0, p_1, \dots, p_d \rangle$ ($1 \leq p_i \leq M$, for all i , M is the fanout of R -tree node), where level- d corresponds to a leaf. For example, the path of the tuple t_1 is $\langle 1, 1, 1 \rangle$. To compute all signatures in a cuboid (e.g., cuboid A), we group tuples by A . Figure 2.b shows a sub-list of tuples with $A = a_1$. For each sub-list, we compute a signature as follows: (1) sort the tuples according to p_0 , (2) scan the sorted list, and set those p_0 bits to 1 in the root bit-array of the signature, and (3) for each sub-sub-list of tuples that share the same value of p_0 , sort it on p_1 , and insert the distinct p_1 to the first child node (i.e., another bit-array) of the root. This procedure is recursively called until the whole signature is completely built. An example is shown in Figure 2.b.

We also assign a path for a node, such that an l -level node

corresponds to a path $\langle p_0, p_1, \dots, p_{l-1} \rangle$. For simplicity, we one-to-one map a path to a *signature ID* (e.g., *SID*) as $SID = p_0 \times (M+1)^l + p_1 \times (M+1)^{l-1} + \dots + p_{l-1}$. In our example, $M = 2$ and the path of the node N_3 is $\langle 1, 1 \rangle$. Its *SID* is 4.

Compressing and Decomposing Signature: We use typical bitmap compression methods [17], [18] to compress signatures. Basically, one can compress the entire signature tree, or compress the bit array in each node individually, and then assemble them to a binary string. In this paper, we adopt the second approach for the following reasons: (1) There is a large room for compression at the node level. For example, with page size $4KB$, the value of M in R -tree node varies from 204 (for two dimensions) to 94 (for five dimensions) [15]; (2) bit arrays in different nodes may have significantly different characteristics, and one may achieve better compression ratio by adaptively choosing different compression scheme; and (3) node-level compression is good for efficient online computation since only the requested nodes need to be decompressed.

We now describe how to decompose the compressed signature into smaller *partial signatures* such that each partial signature fits in a data page with size P . Given a signature tree, we start from the root node and conduct the bread-first traversal. At the same time, we keep track of the accumulated size of traversed nodes. If the size reaches P , we stop the traversal and the first partial signature is generated. Next, we start from the first child N_1 of the root, and conduct bread-first traversal within the subtree under N_1 . Nodes coded by previous partial signatures will be skipped. After finishing the first child, we continue on the following children of the root. If there are still nodes left after the second-level encoding, we will go to the third level, and so on. Each partial signature corresponds to a sub-tree, and is referenced by the cell ID and the *SID* of the root of that sub-tree.

We demonstrate the above algorithm using Figure 2.a as example. Starting from the root node, we generate the first partial signature which contains the root node (10) and the second-level node (11). This partial signature is referenced by the root *SID* ($= 0$). We then start from the root's first child node N_1 . The N_1 node has been coded and is skipped. The two leaf nodes N_3 and N_4 are included. This partial signature is referenced by node N_1 , whose *SID* = 1.

2) *Signature Retrieval:* All signatures are stored on disk and indexed by the cell ID (i.e., $A = a_1$) and the root (of the sub-tree) *SID*. During query processing, we load the partial signatures p only if the node encoded within p is requested. To begin with, we load the first partial signature referenced by the R -tree root. When the query processing model requests a node n which is not presented in the current signature, we use the first level node in the path from the root to n as reference to load the next partial signature. If the partial signature has already been loaded, we check the second-level node in the path, and so on. For example, in Figure 2.a and Figure 1, suppose the a bit in the leaf node N_4 is requested but not presented in the current signature. The path of N_4 is $\langle 1, 2 \rangle$, and we load the partial signature referenced by *SID* = 1 (i.e.,

node N_1).

Due to the curse of dimensionality, we may only compute a subset of low dimensional cuboids, as suggested by [19], [12]. In this case, the P -Cube may not have the signature ready for an arbitrary boolean predicate BP . To assemble a signature for BP online, we need two operators: *signature union* and *signature intersection*. Intuitively, given two signatures s_1 and s_2 , the union operator computes the bit-or result and the intersection operator computes the bit-and result. Suppose the signature to be assembled is s . Any bit that is 1 in s_1 or s_2 will be 1 in s by the union operator. The intersection operator is defined in a recursive way. For each bit b in s , if either the corresponding bit in s_1 or s_2 is 0, the bit is 0. Otherwise, we will examine the intersection result of b 's child nodes. If the intersection causes all bits in the child nodes being 0's, we will set b to 0. An example of signature assembling based on Table I is shown in Figure 3. To assemble signature for arbitrary BP , we assume that the P -Cube always contains a set of *atomic* cuboids (i.e., one-dimensional cuboids on each boolean dimensions).

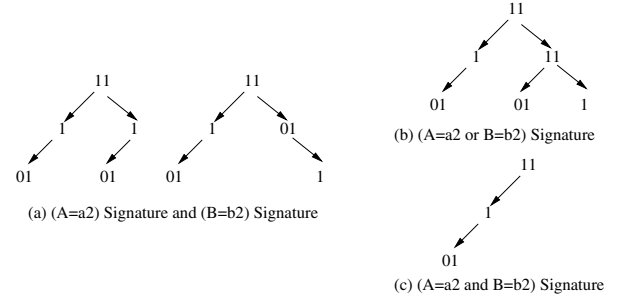


Fig. 3. Assembling signatures

3) *Incremental Maintenance:* Finally, we discuss incremental updates for signatures. We take insertion as an example since the processing for deletion and update is similar. Inserting a new tuple to R -tree may cause node splitting and tuple re-insertion [15], [16]. Before presenting the complete solution, we first discuss a simpler case where there is no node splitting nor tuple re-insertion.

Every node (including leaf) in R -tree can hold up to M entries. We assume each node keeps track of its free entries. When a new tuple is added, the first free entry is assigned. In case there is no node splitting nor tuple re-insertion, only the path of the newly inserted tuple is updated, and those for other tuples keep the same.

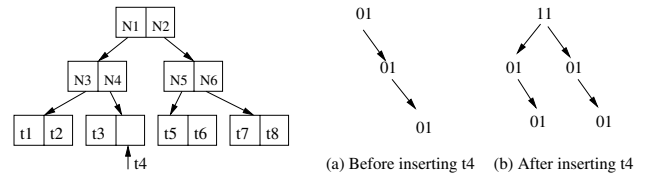


Fig. 4. Insertion without Node splitting

Suppose the database already has $t_1, t_2, t_3, t_5, \dots, t_8$, a new

tuple t_4 is inserted. Figure 4 demonstrates the change of ($A = a_3$) signature (see Table I). t_4 is first inserted into R -tree and finds an entry in leaf node N_4 . A new $path = \langle 1, 2, 2 \rangle$ is computed for t_4 . Since the $paths$ for all other tuples did not change, only the signatures of cells a_3 , b_3 and a_3b_3 in cuboids A , B , and AB are affected. Furthermore, only the entries on the path from the root to t_4 are possibly affected. We then load those partial signatures containing the $path$, and flip the corresponding entries from 0 to 1.

When a node is split, the $paths$ of all tuples under the split entry will change. To correctly update signatures, we need to collect the old and new $paths$ for those tuples. We first traverse the sub-tree under the entry before splitting to get old $paths$, and then traverse it again after splitting to get new $paths$. Similarly, for tuples scheduled for re-insertion, we also compute the old and new $paths$. As a result, there is a set of tuples whose $paths$ are changed, and all cells corresponding to those tuples need to be updated.

V. QUERY PROCESSING USING P-CUBE

Having presented the *signature* measure and *P-Cube*, we discuss how to use signatures in query processing. We first describe how to answer skyline queries, and then extend it to top- k queries. Finally, we describe the execution of roll-up and drill-down queries.

A. Processing Skyline Queries

To begin with, we outline the *signature-based* query processing in Algorithm 1. The algorithm follows the branch-and-bound principle to progressively retrieve data nodes [9]. We briefly explain each step as follows. Line 1 initializes a list to store the final results. Each node n is associated with a value $d(n)$, where $d(n) = \min_{x \in n} (\sum_{i=1}^j N'_i(x))$ is the lower bound value over the region covered by n , and $N'_i(x)$ is the value of object x on preference dimension N'_i . Clearly, a data object t cannot be dominated by any data objects contained by node n if $\sum_{i=1}^j N'_i(t) \leq d(n)$ [9]. On the other hand, if a node n is dominated by some data object t , all child nodes of n are dominated by t . The root of the c_heap contains an entry e with minimal $d(e)$ (lines 2-4). Line 5 checks whether e is pruned by the boolean predicate, or dominated by previous skylines stored in $result$. For every e that passes the checking, e is a new skyline if it is a data object (lines 7-8). Otherwise, the algorithm further examines e 's child nodes (lines 10-12).

The *prune* procedure has two main tasks: *domination pruning* and *boolean pruning*. The domination pruning for skylines are straightforward: simply comparing the value in each dimension between the submitted entry e and the previously discovered skylines. If e is dominated by any of them, e can be pruned. The boolean pruning is accomplished by signatures. No matter whether the entry e submitted to the *prune* procedure is a data tuple or an intermediate node, we can always get the $path$ of e , and query the signature using the $path$. The *prune* procedure also maintains two optional global lists: the b_list and d_list . The b_list keeps all the entries pruned by boolean predicates, and the d_list keeps all

Algorithm 1 Framework for Query Processing

Input: R -tree R , P -Cube C , user query Q

```

1:  $result = \phi$ ; // initialize the result set
2:  $c\_heap = \{R.root\}$ ; // initialize candidate heap
3: while ( $c\_heap \neq \phi$ )
4:   remove top entry  $e$ ;
5:   if ( $prune(e)$ )
6:     continue;
7:   if ( $e$  is a data object)
8:     insert  $e$  into  $result$ ;
9:   else //  $e$  is a node
10:    for each child  $e_i$  of  $e$  // expand the node
11:      if ( $\neg prune(e_i)$ )
12:        insert  $e_i$  into  $c\_heap$ ;
13: return
```

Procedure *prune*(e)

```

Global Lists:  $b\_list, d\_list$ ;
14: if ( $preference\_prune(e) == false$ )
15:   insert  $e$  into  $d\_list$ ;
16:   return true;
17: if ( $boolean\_prune(e) == false$ )
18:   insert  $e$  into  $b\_list$ ;
19:   return true;
20: return false;
```

the entries pruned by skyline domination. The sole purpose of maintaining these lists is to efficiently support roll-up/drill-down queries (Section V-C).

One can easily verify the correctness of the algorithm for skyline queries. The results are generated on Line 8, and we shall show that all data objects e on Line 8 are valid skylines. This is true because: (1) e passes the boolean pruning, and thus e satisfies the boolean predicate; and (2) e passes the preference pruning, and thus e is not dominated by previous skylines. Since e appears at the top of the c_heap , and thus e can not be dominated by any future data objects. These two facts ensure that e is a valid skyline.

Here we estimate the overall I/O cost of Algorithm 1 for skyline queries. The cost consists of two parts: C_{sig} and C_{R-tree} . The former represents the cost for loading signatures and the latter is the cost for loading R -tree blocks. In reality, one partial signature encodes many R -tree nodes. For example, a partial signature generally has size $4KB$ (i.e., a page size), and the size of each bit-array in the signature is up bounded by $\frac{M}{8}$ bytes (without compression). Here M is the maximal number of child entries in the node. For a 2 dimensional R -tree, $M = 204$ and a partial signature can encode 160 nodes. Thus, $C_{sig} \ll C_{R-tree}$, and we focus on C_{R-tree} only. When $BP = \phi$ (i.e., no boolean predicates), Papadias *et al.* [9] show that the progressive query framework demonstrated in Algorithm 1 is I/O optimal such that the algorithm only retrieves R -tree blocks that may contain skylines. When $BP \neq$

ϕ , since the signature provides exact answers for boolean checking, Algorithm 1 only retrieves R -tree blocks that pass domination checking and boolean checking. We have the following claim.

Lemma 1: For skyline queries with boolean predicates, Algorithm 1 is optimal in terms of $C_{R\text{-tree}}$ such that the number of R -tree blocks retrieved is minimized.

B. Processing Top- K Queries

We discuss how to process top- k queries using Algorithm 1. Comparing to the skyline query processing, top- k query processing differs in two ways: (1) how the nodes are maintained by the candidate heap (i.e., c_heap); and (2) how the *prune* procedure (i.e., lines 14 to 20) works.

To push preference pruning deep into the database search, the nodes have to be scheduled in the best-first way. That is, nodes which are most promising for top scores need to be accessed first. Suppose the ranking function is f . We define $f(n) = \min_{x \in n} f(x)$, for each node n . Consequently, the candidate heap uses $f(n)$ to order nodes, and the root has the minimal score.

In the *prune* procedure, the signature-based boolean pruning is the same as that in skyline queries. The preference pruning consists of two steps. First, for each candidate e (node or data object) submitted for prune checking, the preference pruning will first compare $f(e)$ with all data objects in the *result* (line 2) list. If there are at least k objects whose scores are better than $f(e)$, then e can be pruned. Second, at any time, we can sort data objects in the *result* list according to their f scores, and only need to keep top- k in the *result* list. Similar to Lemma 1, in top- k query processing, the number of R -tree blocks that are retrieved by Algorithm 1 is optimal.

C. Drill Down and Roll up Queries

Drill-down and roll-up are typical OLAP operators applied on boolean dimensions. Given the current boolean predicate BP , the drill-down query strengthens BP by augmenting an additional boolean predicate, and roll-up query relaxes BP by removing some boolean predicate in BP . For example, let $BP = \{A = a_1, B = b_1\}$. $BP' = \{A = a_1, B = b_1, C = c_1\}$ is a drill-down, whereas $BP' = \{A = a_1\}$ is a roll-up. Drill-down and roll-up queries are incremental in that they always follow a standard query. We demonstrate our method by skyline queries.

A standard skyline query starts with an empty c_heap and searches from the root node. While in drill-down and roll-up queries, we can avoid searching from scratch. Thus, the I/O cost may be reduced. Recall that in Algorithm 1, we maintain three lists: *result*, *b_list*, and *d_list*, where *result* contains the results for current query, *d_list* contains entries dominated by objects in *result*, and *b_list* contains entries not satisfying boolean predicates. Lemma 2 shows how to re-construct c_heap without starting from the root node.

Lemma 2: Suppose *result*, *b_list*, and *d_list* are maintained by the last query. For a continuing drill-down (or roll-up) query, re-constructing the candidate heap as $c_heap =$

$result \cup d_list$ (or $c_heap = result \cup b_list$) returns the correct answers.

With the re-constructed c_heap , we execute Algorithm 1 from Line 4. Note the size of c_heap can be further reduced by enforcing boolean checking and domination checking beforehand. Using drill-down as example, for each previous skyline object in *result*, it continues to be a skyline objects if it satisfies the drill-down boolean predicate. Otherwise, it is directly moved to *b_list*. For each entry in *d_list*, we filter it by the *prune* procedure before we insert it into c_heap .

VI. PERFORMANCE STUDY

A. Experimental Setting

We use both synthetic and real data sets. The real data set we consider is the *Forest CoverType* data set obtained from the UCI machine learning repository web-site (www.ics.uci.edu/~mllearn). This data set contains 581,012 data points with 54 attributes. We select 3 quantitative attributes (with cardinalities 1989, 5787, and 5827) as preference dimensions, and other 12 attributes (with cardinalities 255, 207, 185, 67, 7, 2, 2, 2, 2, 2, 2, 2) as boolean dimensions. We also generate a number of synthetic data sets for our experiments. For each synthetic data, \mathcal{D}_p denotes the number of preference dimensions, \mathcal{D}_b the number of boolean dimensions, \mathcal{C} the cardinality of each boolean dimension, \mathcal{T} the number of tuples.

We build all *atomic* cuboids (i.e., all single dimensional cuboids on boolean dimensions) for *P-Cube*. Signatures are compressed, decomposed and indexed (using B_+ -tree) by cell IDs and *SID*'s. The page size in R -tree is set as $4KB$. We compare our proposed signature-based approach (referred as *Signature*) against the following alternative methods.

Boolean first (referred as *Boolean*): We use B_+ -tree to index each boolean dimension. Given the boolean predicates, we first select tuples satisfying the boolean conditions. This may be conducted by index scan or table scan, and we report the best performance of the two alternatives. We then use Algorithm 1 to compute the skylines or top- k results (by removing the boolean prune components on lines 17-19).

Domination first (referred as *Domination* for skyline queries and *Ranking* for top- k queries): We combine the BBS algorithm [9] and minimal probing method [3]. Both are the state-of-art methods in the literature. The BBS algorithm is similar to Algorithm 1, except that there is no boolean checking in the *prune* procedure. For each candidate result, we conduct a boolean verification guided by the minimal probing principle [3]. The boolean verification involves randomly accessing data by *tid* stored in the R -tree, and we only issue a boolean checking for a tuple in between lines 7 and 8 in Algorithm 1. In doing this, the number of boolean verification is *minimized*[3].

Index merge: This is for top- k queries only. We use the progressive and selective index merge framework proposed by [14]. More specifically, we build B_+ -tree indices on boolean

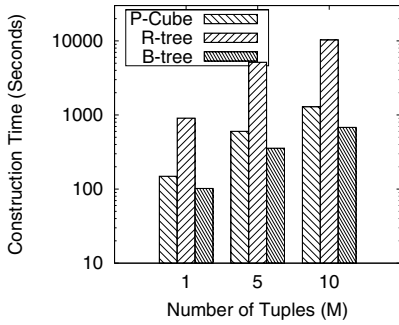


Fig. 5. Construction Time w.r.t. \mathcal{T}

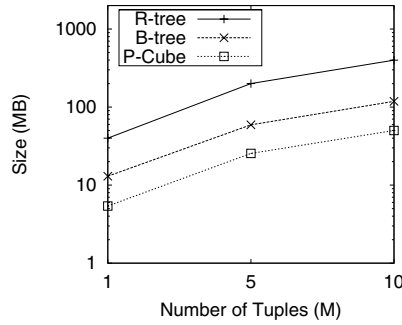


Fig. 6. Materialized Size w.r.t. \mathcal{T}

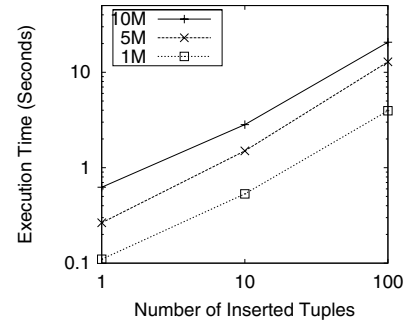


Fig. 7. Incremental Update Time

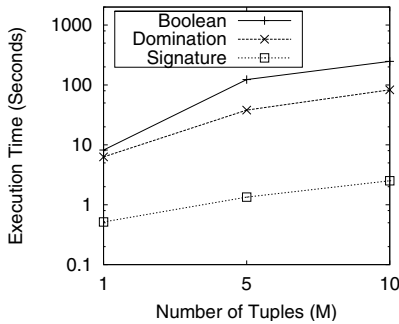


Fig. 8. Execution Time w.r.t. \mathcal{T}

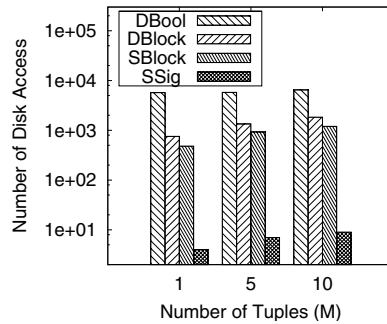


Fig. 9. Number of Disk Access w.r.t. \mathcal{T}

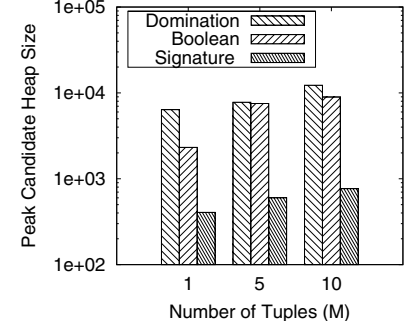


Fig. 10. Peak Candidate Heap Size w.r.t. \mathcal{T}

dimensions, and R -tree index on preference dimensions. Given a query with boolean predicates, we join all corresponding indices. The ranking function is re-formulated as follows: if a data satisfies boolean predicates, the function value on preference dimensions is returned. Otherwise, it returns MAX value (assuming minimal top- k).

B. Experimental Results

Experiments are conducted to examine the construction and space costs, and the query performance.

1) *Construction and Space Costs for P-Cube*: We first examine the cost to construct, store, and incrementally maintain P -Cube. We use synthetic data set in this set of experiments. By default, both the number of boolean dimension D_b and the number of preference dimension D_p are 3. The cardinality \mathcal{C} of each boolean dimension is 100, and the distribution \mathcal{S} among preference dimensions is uniform.

To study the scalability, we vary \mathcal{T} (the number of tuples) from 1M to 10M. Figure 5 shows the costs to build R -tree partition, compute P -Cube, and build B_+ -tree index for all boolean dimensions. The R -tree is shared by both *Signature* and *Domination* approaches and B_+ -trees are used by *Boolean* method. We observe that the computation of P -Cube is 7-8 times faster than that of R -tree, and is comparable to that of B_+ -tree. On the other hand, for space consumption, P -Cube is 2 times less than B_+ -trees and 8 times less than R -tree (Figure 6).

To examine the performance of incremental update of P -Cube, we insert 1 to 100 new tuples. The execution times in Figure 7 show that the incremental maintenance algorithms are much better than the re-computation since we only update target cells. Moreover, maintenance in batch is more scalable than maintenance tuple by tuple. For example, the execution time for inserting one tuple in 1M data is 0.11 Seconds. While the average cost for inserting 100 tuple on the same data is 0.04 Seconds.

2) *Performance on Skyline Queries*: We start to evaluate the on-line query behavior from this subsection. To begin with, we examine the query performance on skylines, with single selection condition. The results on multiple boolean predicates are reported in Section VI-B.4.

We run skyline queries on the same synthetic data sets as those in Figure 5, and the execution time is shown in Figure 8. We compare *Signature* with *Boolean* and *Domination*. Clearly, the signature-based query processing is at least one order of magnitude faster. This is because in *Boolean*, disk access is based on boolean predicates only, and in *Domination*, disk access solely relies on domination analysis. *Signature* combines both pruning opportunities and thus avoids unnecessary disk accesses.

To take a closer look, we compare the number of disk accesses between *Signature* and *Domination* in Figure 9. *Domination* consists of two types of disk accesses: R -tree block retrieval (*DBlock*) and random tuple access for boolean verification (*DBoo*). *Signature* also consists of two types of

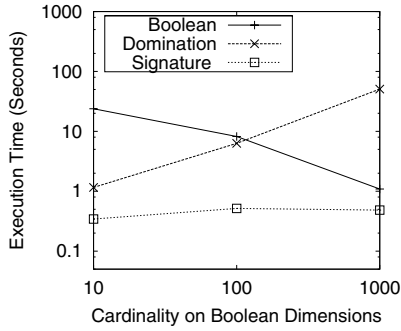


Fig. 11. Execution Time w.r.t. C

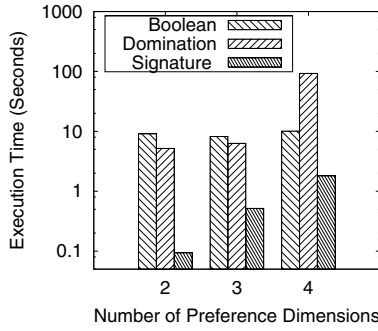


Fig. 12. Execution Time w.r.t. D_p

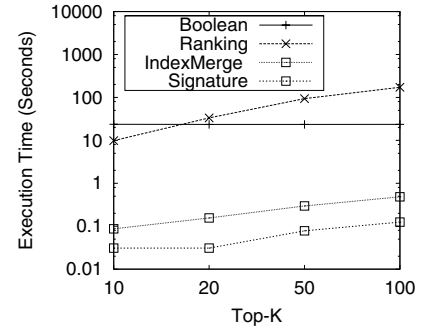


Fig. 13. Execution Time w.r.t. k

disk access: signature loading (*SSig*) and *R*-tree block retrieval (*SBlock*). We observe that (1) in *Signature*, the cost of loading signature is far smaller ($\leq 1\%$) than that of retrieving *R*-tree blocks, and (2) guided by the signatures, our method prunes more than 1/3 *R*-tree blocks comparing with *Domination* and avoids even more random tuple accesses for boolean verification.

From another perspective, reducing the memory requirement is equally important for the scalability issue. Note this is not necessarily implied by minimizing disk accesses. For example, we adopt a lazy verification strategy in *Domination* and this has trade-off of keeping more candidates in heap. Figure 10 compares the peak size of candidate heap in memory for all three methods. With *Signature*, the number of entries kept in memory is an order of magnitude less than that of *Domination* and *Boolean*.

The boolean (preference) selectivity determines the filtering power by boolean predicates (multi-dimensional dominations). We vary the cardinality C of each boolean dimension from 10 to 1000, while keeping $T = 1M$. The query execution time is shown in Figure 11. As expected, *Boolean* performs better when C increases and the performance of *Domination* deteriorates. The preference selectivity is affected by the number of preference dimensions. We generate a sets of synthetic data with the number of preference dimension varying from 2 to 4. The query performance is shown in Figures 12. It becomes more challenging to compute the skyline results when the number of dimension goes high, and the computation time for *Domination* increases. On the other hand, the preference selectivity has limited effect on *Boolean*. In all experiments, *Signature* performs fairly robustly and is consistently the best among the three.

3) *Performance on Top-K Queries*: We continue to evaluate the performance on top- k queries. The algorithms have similar behaviors as those in skyline queries in many tests, such as scalability and boolean selectivity. In this subsection, we further conduct experiments on query performance with respect to the value of k .

Suppose the ranking function is formulated on three attributes X , Y , and Z , and they are partitioned by an *R*-tree. For demonstration, we use a *linear* query with function $f_l = aX + bY + cZ$, where a , b , and c are random parameters.

We use the $1M$ synthetic data in Figure 8. The query execution time on linear function with respect to different k values in Figure 13. We observe that *Boolean* is not sensitive to the value of k ; and *Ranking* performs better when k is small. *Signature* runs order of magnitudes faster, and it also outperforms *Index Merge*. This is because *Index Merge* joins the search space online, while the signature materializes the joint space offline. Signature leverages the property that the space of boolean selection is fixed.

4) *Performance with Boolean Predicates*: The last group of experiments evaluates the query performance w.r.t. boolean predicates. We use the real data set *Forest CoverType*, which consists of 12 boolean dimensions and 3 preference dimensions.

We issue skyline queries with 1 to 4 boolean predicates and the execution time is shown in Figure 14. *Signature* and *Boolean* are not sensitive to boolean predicates, and the former performs consistently better. *Domination* requests more boolean verification, and thus the execution time grows significantly. When there are $k > 1$ boolean predicates, we essentially need to load k one-dimensional signatures since only atomic cuboids are materialized. The comparison of execution time used by signature loading and query processing is shown in Figure 15. The time used for loading signatures increases slightly with k . However, even when there are 4 boolean predicates, the signature loading time is still far less than the query processing time (*i.e.*, less than 10%). Figure 15 suggests that materializing atomic cuboids only may be good enough in real applications.

Figure 16 shows the performance gains of drill-down queries over new queries. The performance for roll-up query is similar. For each query with k ($k \geq 2$) boolean predicates in Figure 14, the drill-down query is executed in two steps: (1) submit a query with $k - 1$ boolean predicates, and (2) submit a drill-down query with the additional k^{th} boolean predicate. We observe more than 10 times speed-up by caching the previous intermediate results and re-constructing candidate heap upon them.

VII. DISCUSSION AND CONCLUSIONS

Besides the lossless compression discussed in this paper, lossy compression such as Bloom Filter [20] is also applicable.

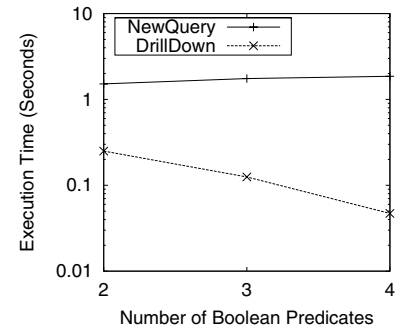
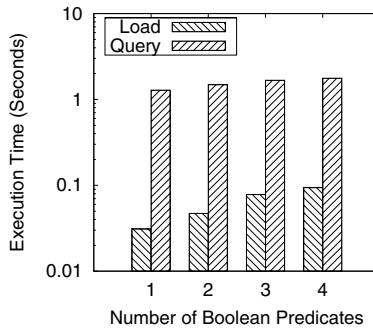
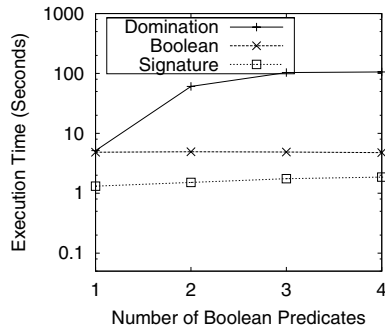


Fig. 14. Execution Time w.r.t. Boolean Predicates

Fig. 15. Signature Loading Time vs. Query Time

Fig. 16. Drill-Down Query vs. New Query

We can build a bloom filter on all *SID*'s whose corresponding entries are 1 in the signature. During query execution, we can load the compressed signature (i.e., a bloom filter), and test a *SID* upon that. Algorithm 1 can also be easily extended to support other preference queries, such as dynamic skyline queries [9] and convex hull queries [21].

In conclusion, we develop a signature-based approach that seamlessly integrates both boolean pruning and preference pruning. Our performance evaluation shows that the proposed method improves previous approaches by at least one order of magnitude.

VIII. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their invaluable feedback. This research was supported in part by the U.S. National Science Foundation (NSF) IIS-05-13678/06-42771 and BDI-05-15813. The work was done when Dong Xin worked at the University of Illinois at Urbana-Champaign.

REFERENCES

- [1] J. Chomicki, "Preference queries," *CoRR*, vol. cs.DB/0207093, 2002.
- [2] S. Borzsonyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.
- [3] K. C.-C. Chang and S. won Hwang, "Minimal probing: Supporting expensive predicates for top-k queries," in *SIGMOD Conference*, 2002, pp. 346–357.
- [4] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001.
- [5] M. J. Carey and D. Kossmann, "On saying "Enough already!" in SQL," in *SIGMOD Conference*, 1997, pp. 219–230.
- [6] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid, "Rank-aware query optimization," in *SIGMOD Conference*, 2004, pp. 203–214.
- [7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *ICDE*, 2003, pp. 717–816.
- [8] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries," in *VLDB*, 2002, pp. 275–286.
- [9] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 41–82, 2005.
- [10] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *VLDB*, 2001, pp. 301–310.
- [11] S. Churdhuri and U. Dayal, "An overview of data warehousing and data cube," *SIGMOD Record*, vol. 26, pp. 65–74, 1997.
- [12] D. Xin, J. Han, H. Cheng, and X. Li, "Answering top-k queries with multi-dimensional selections: The ranking cube approach," in *VLDB*, 2006, pp. 463–475.

- [13] Z. Zhang, S.-W. 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*, 2006, pp. 359–370.
- [14] D. Xin, J. Han, and K. C.-C. Chang, "Progressive and selective merge: Computing top-k with ad-hoc ranking functions ." in *SIGMOD Conference*, 2007, pp. 103–114.
- [15] A. Guttman, "R-tree: A dynamic index structure for spatial searching," in *SIGMOD Conference*, Boston, MA, June 1984, pp. 47–57.
- [16] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD Conference*, 1990, pp. 322–331.
- [17] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*. Prentice Hall, 2002.
- [18] A. Fraenkel and S. Klein, "Novel compression of sparse bit-strings - preliminary report," *Combinatorial Alg. on Words, NATO ASI Series*, vol. 12, pp. 169–183, 1985.
- [19] X. Li, J. Han, and H. Gonzalez, "High-dimensional olap: A minimal cubing approach," in *VLDB*, 2004, pp. 528–539.
- [20] B. H. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Comm. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
- [21] C. Bohm and H.-P. Kriegel, "Determining the convex hull in large multidimensional databases," in *DaWaK*. Springer-Verlag, 2001, pp. 294–306.