BITPEER: Continuous Subspace Skyline Computation with Distributed Bitmap Indexes

Katerina Fotiadou
Computer Science Department
University of Ioannina, Greece
kfotiado@cs.uoi.gr

Evaggelia Pitoura
Computer Science Department
University of Ioannina, Greece
pitoura@cs.uoi.gr

ABSTRACT

In this paper, we propose a bitmap approach for efficient subspace skyline computation in a distributed setting. Our approach computes extended skylines which have been shown to include all points necessary for computing the skyline at any subspace. We present an algorithm for computing extended skylines using a bitmap representation along with a storage efficient bucket-based variation of this representation. We provide a caching scheme so that subspace skyline queries can re-use the results of previously computed similar queries. We also introduce a method for grouping continuous subspace queries for supporting their efficient computation. Finally, we present preliminary experimental results of the performance of our approach.

1. INTRODUCTION

Peer-to-peer systems have attracted a lot of attention as a means of data sharing and exchange amongst dynamic populations of users. In such settings, it is central that users are equipped with powerful query languages that would allow them to locate the data items that are of interest to them. In this paper, we focus on supporting skyline queries.

Skyline queries return those data points in a multidimensional dataset that are not dominated by any other point in the set [2]. A data point p_1 *dominates* some other data point p_2, if it is not worse than p_2 in any dimension and it is better than p_2 in at least one dimension. *Subspace skyline queries* consider dominance in subspaces of the dimensions of the dataset. Such queries are appealing, since they allow users to locate the best points along the dimensions that are of interest them. For instance, a user looking for movies may pose skyline queries on selected dimensions, for example, on the duration, year of production or language dimensions, as long as, the domain of each dimension is ordered (either, naturally or according to user preferences).

We assume that the dataset is distributed among a set of nodes (peers). Subspace skyline queries may be posed at any peer and are computed over the whole dataset. In particular, each peer computes the skyline of its local dataset. Then, the peers exchange their skyline points, so that the overall skyline points are computed. This way peers do not need to communicate their whole dataset which is important from both a privacy and an efficiency perspective. To better coordinate the execution of skyline queries, peers form groups co-ordinated by some peer in the group, called a *superpeer*. The superpeer maintains the skyline points of the datasets of its group members.

Many approaches have been proposed for the efficient computation of both skylines and subspace skylines queries. We adopt a Bitmap approach [10] because of (a) the small size of the representation of the dataset that results in reduced storage overheads and more importantly in reduced communication overheads for its transmission among the peers, (b) the fast bitwise computation it provides and (c) its support for progressive evaluation, so that, it is cost-efficient to check whether a new data point belongs to the skyline.

For subspace skyline computation, the main challenge lies in achieving computation sharing among the skylines of different subspaces, since, in general, there is no subset relationship between the skylines of subset subspaces [15]. We compute *extended skylines* [11] which contain all data points that are necessary for computing any subspace skyline. We show how to efficiently compute extended skylines using bitmaps. It turns out that extended skylines work also well with a storage efficient bucket-based variation of bitmaps.

An important performance issue is re-using previous computations. To this end, we exploit skyline caching. Peers cache the results of previous subspace skyline queries and re-use them when a similar query is issued. By caching extended skyline points, results can be re-used not only by the same but also by similar queries. We also consider continuous queries, that is, queries that are executed more than once, either periodically or when there is some change in the dataset that affects their result. Continuous skyline queries are very useful, since they allow users to monitor the dataset and get informed about new interesting points, such as a more interesting movie or a better priced air-ticket. We propose an approach for grouping continuous queries, to avoid computing each one of them separately.

In a nutshell, in this paper, we:

- propose a bitmap approach for computing extended skyline queries along with a cost-efficient bucket variation,
- present a caching scheme for subspace skylines in peer-to-peer systems, and

- introduce a method for grouping continuous subspace skyline queries for efficiency.

The remainder of this paper is structured as follows. In Section 2, we introduce the basic bitmap-based algorithm for subspace skyline computation. In Section 3, we present its deployment in a peer-to-peer setting and the related algorithms. In Section 4, we discuss caching and in Section 5 continuous subspace skyline evaluation. In Section 6, we report our experimental results. Finally, in Section 7, we compare our work with related research, while in Section 8, we offer our conclusions.

2. BITMAP-BASED SUBSPACE SKYLINE COMPUTATION

2.1 Background

Assume a space (i.e. dimension set) D defined by a set of d dimensions $\{d_1, d_2, \ldots, d_d\}$. Let S be a dataset of data points on D. For each point $p \in S$, we use $p(i)$ to denote the value of point p in dimension d_i.

Informally, a point p dominates a point q, if p is not worse than q in any dimension and it is better than q in at least one dimension. More formally, given a dataset S on space $D = \{d_1, d_2, \ldots, d_d\}$, the *dominance relation*, $\succ_d$, defines a partial order on the points in S, such that $p \in S$ dominates $q \in S$, $p \succ_d q$, if and only if (i) $\forall\, i,\ 1 \leq i \leq d$, $p(i) \geq q(i)$ and (ii) $\exists\ j,\ 1 \leq j \leq d$, such that $p(j) > q(j)$. The dominance relation is non-reflexive, antisymmetric and transitive.

The *skyline* of a dataset S is the set of points in S that are not dominated by any other point in S.

2.1.1 Bitmap Representation

Many algorithms have been proposed for computing the skyline of a set of data points. In this paper, we use the bitmap representation introduced in [10].

Assume that each point p can take k_i distinct values for each dimension i. We use v_{ij} to represent the j-th value of the i-th dimension, where $v_{i_1} > \ldots > v_{i_{k_i}}$. Each point p is represented by an m-bit vector, with k_i bits assigned to each value $p(i)$, thus $m = \sum_{j=1}^{d} k_j$. The first k_1 bits are assigned to $p(1)$, the next k_2 to $p(2)$ and so on. Let the j-th bit corresponds to value v_{ij}. If $p(d_i)$ is the v_{il}-th value of d_i, then the k_i bits of $p(d_i)$ are set as follows, the bits at positions 1 to $l-1$ are set to 0 and the bits at positions l to k_i are set to 1.

An example is shown in Fig. 1. There are 4 points with $d = 3$ dimensions. Dimension d_1 has four distinct values (namely, 4, 3, 2, 1), d_2 has 3 distinct values (namely, 3, 2, 1) and d_3 has 2 distinct values (namely, 2, 1). For instance, point (1, 1, 2) is represented by the bit vector (000100111).

The array of vectors for representing the data points in the dataset S is stored column-wise as an array of bit slices. Let BS_{ij} represent the bit-slice for the jth distinct value of dimension d_i. Assume that the value $p(d_i)$ corresponds to value $u_{i_{m_i}}$ of d_i. To check whether a point p belongs to the skyline, we perform the following steps:

1. We compute the bitslice A as $A = BS_{1_{m_1}}\ AND\ BS_{2_{m_2}}\ AND\ \ldots\ AND\ BS_{d_{m_d}}$, where AND stands for the bitwise *and* operation. Note that the nth bit in A is 1 if only if the nth point is S has value in each dimension greater or equal of the value of p in this dimension.

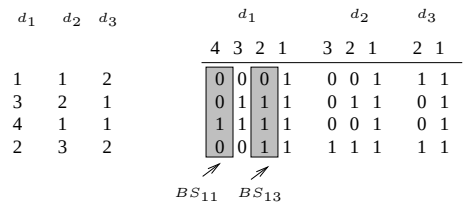

Figure 1: Data points and the corresponding bitmap structures

2. We compute the bitslice B as $B = BS_{1_{m_1-1}}\ OR\ BS_{2_{m_2-1}}\ OR\ \ldots\ OR\ BS_{d_{m_d-1}}$, where OR stands for the bitwise *or* operation and $BS_{i0} = 0$ for all i. Note that the nth bit in B is 1 if and only if the nth point in S has value in some dimension that is greater than the corresponding value of p.

3. Finally, let $C = A\ AND\ B$. Point p belongs to the skyline of S if and only if all bits of C are 0.

For example, for point (3, 2, 1) in Fig. 1, $A = BS_{12}\ AND\ BS_{22}\ AND\ BS_{32} = 0110\ AND\ 0101\ AND\ 1111 = 0100$, $B = BS_{11}\ AND\ BS_{21}\ AND\ BS_{31} = 0010\ OR\ 0001\ OR\ 1001 = 1011$, and $C = A\ AND\ B = 0000$, which means that (3, 2, 1) belongs to the skyline. Whereas for point (1, 1, 2), $A = 1111\ AND\ 1111\ AND\ 1001 = 1001$, $B = 0111\ OR\ 0101\ OR\ 0000 = 0111$, and $C = 0001$, which means that (1, 1, 2) does not belong to the skyline.

2.1.2 Subspace Skyline

Each non-empty subset U of D, $U \subseteq D$, is called a *subspace*. We denote with p_U the projection of point $p \in S$ to U. Specifically, p_U is the tuple $(p(d_{i_1}), p(d_{i_2}), \ldots, p(d_{i_{|U|}}))$ where $d_{i_1}, d_{i_2}, \ldots\ d_{i_{|U|}} \in U$. The skyline of a set of data points S in the subspace U, $SKY_U(S)$, is the skyline of the projection of these points in this subspace. Specifically, the projection of a point p ($p \in S$) in $U \subseteq D$ belongs to the skyline of U, if p_U is not dominated by any point $q_U \in U$ for all $q \in S$. There are $2^d - 1$ possible subspace skyline queries.

It turns out that in general, for two subspaces U and V, $U \subset V$, there is no containment relationship between their skylines as the following theorem states [15]:

THEOREM 1. *Let S be a dataset of d-dimensional points in D. Let two subspaces $U \subseteq D$ and $V \subseteq D$ where $U \subset V$. In subspace V, for each point q that belongs to the skyline of subspace U it holds:*

- *either q is dominated in V by a point $p \in SKY_U(S)$ or*

- *q belongs to the $SKY_V(S)$*

This means that the skyline of U is not necessarily a subset of the skyline of V or vice versa. This happens only if the distinct value condition holds as it is easy to conclude from the theorem above:

COROLLARY 1. *Let S be a dataset of d-dimensional points in D. If for any two point p and q, $p(d_i) \neq q(d_i)\ \forall\ d_i \in D$, then for any two subspaces $U \subseteq D$ and $V \subseteq D$ where $U \subset V$, it holds $SKY_U(S) \subseteq SKY_V(S)$.*

2.1.3 Extended Skyline

For efficiently computing subspace skyline queries, the ext-dominance relation is proposed in [11]. For each subspace U, a point p *ext-dominates* q if for each dimension d_i of U, $p(d_i) > q(d_i)$. The extended skyline of U is the set of data points that are not ext-dominated by any other point. In other words, for two points in the extended skyline either (skyline) dominance holds or they have the same value in at least one dimension. The following two properties hold [11]:

PROPERTY 1. *Each point that belongs to the skyline of U also belongs to the extended skyline of U, $SKY_U \subseteq$ ext-SKY_U.*

PROPERTY 2. *Each point that belongs to the skyline of a subspace V, $V \subseteq U$, also belongs to the extended skyline of U, $SKY_V \subseteq$ ext-SKY_U.*

That is, the extended skyline of all dimensions is such that the skyline of any subspace can be computed from it.

2.2 Bitmap-based Extended Skyline Computation

The bitmap algorithm can be used to check whether a point p belongs to the skyline of any subspace just by using only the bitslices of the dimensions in this subspace. For example, take the projection of point (3, 2, 1) in Fig. 1 in subspace $U = \{d_1, d_3\}$, that is, point (3, 1). To check whether (3, 1) belongs to the subspace skyline SKY_U, we compute $A = 0110\ AND\ 1111 = 0110$, $B = 0010\ OR\ 1001 = 1011$, and $C = A\ AND\ B = 0010$, which means that (3, 1) does not belong to SKY_U, although it belongs to SKY_D. In fact, point (3, 1) is dominated in U by point (4, 1), i.e., the projection of (4, 1, 1).

Next, we present a bitmap algorithm for computing the extended skyline for a set of data points S. The algorithm uses the same bitmap representation for the points in dataset S as the original algorithm, but it is simpler. In particular, to check whether a point p belongs to the extended skyline, we perform just the following step:

1. We compute the bitslice C as $C = BS_{1_{m_1-1}}\ AND\ BS_{2_{m_2-1}}\ AND \ldots\ AND\ BS_{d_{m_d-1}}$, where AND stands for the bitwise *and* operation and $BS_{i0} = 0$ for all i. Point p belongs to the extended skyline of S if and only if all bits of C are 0.

Referring to our example points in Fig. 1, using the above step to check whether point (1, 1, 2) belongs to the extended skyline, we get $C = 0111\ AND\ 0101\ AND\ 0000 = 0000$. Thus, point (1, 1, 2) belongs to the extended skyline, although it does not belong to the skyline. This is because (1, 1, 2) has the same value in d_3 with point (2, 3, 2) which belongs to the skyline. Point (3, 2, 1) belongs to the skyline, so it should also belong to the extended skyline by Property 2. Indeed, using the above step, we get $C = 0010\ AND\ 0001\ AND\ 1001 = 0000$.

One shortcome of the bitmap approach to computing the (extended) skyline of a dataset S is that we need to check the membership of each point p in S. We describe next a couple of pruning steps for quickly including or excluding groups of points from the extended skyline computation.

Assume that point p is known to belong to the extended skyline. By the definition of the extended skyline, we know that: (a) all points that have a value larger than the corresponding value of p in at least one dimension belong to the extended skyline, and that (b) all points that have values smaller than p in all dimensions do not belong to the extended skyline. We call the points that satisfy (a) definitely extended sklyline (DSK) points and the points that satisfy (b) definitely not extended skyline (DNSK) points.

Let p belong to the extended skyline and value $p(d_i)$ correspond to value $u_{i_{m_i}}$ of d_i. To compute the DSK points, we perform the following test:

> [*DSK test*] We compute the bitslice D as $D = BS_{1_{m_1-1}}\ OR\ BS_{2_{m_2-1}}\ OR \ldots\ OR\ BS_{d_{m_d-1}}$ and $BS_{i0} = 0$ for all i. The nth bit in D is 1 if and only if the nth point in S has value in some dimension that is greater than the corresponding value of p. We include all such points in the extended skyline.

To compute the DNSK points, we perform the following test:

> [*DNSK test*] We compute the bitslice E as $d = BS_{1_{m_1}}\ OR\ BS_{2_{m_2}}\ OR \ldots\ OR\ BS_{d_{m_d}}$. The nth bit in E is 0 if only if the nth point in S has at all dimensions values smaller than the corresponding values of p. We exclude all such points from the extended skyline.

The efficacy of the above tests depends on which point p is used for computing the DSK and DNSK points. In general, we want a point that has a large value in at least one dimension. One way to select such a point is by first selecting a dimension, say d_i. Then, for this dimension we select the point that has value 1 at the largest v_{ij} and use this as point p.

A combination of the above may also be used for computing fast a few extended skyline points as follows:

1. Select the columns that correspond to the largest value for each dimension.
2. Check which points have value 1 at this column. These points belong to the extended skyline and can be returned to the issuer of the query.
3. From those points, select the one that has the most 1s among them. Use this point as point p for computing the DSK and DNSK points.

2.3 Bucket Bitmap Representation

The original bitmap algorithm works well for categorical attributes but it is not directly applicable to dimensions whose domains have continuous values. A simple solution is to divide the domain in ranges, that we call *buckets*. The larger the number of buckets, the more precise the representation of the domain. However, a large number of buckets increases the size of the bitmap representation of the points in the dataset.

The main problem with the bucket representation approach is that it may introduce *false negatives*, that is, there may be points that belong to the skyline that the bitmap algorithm fails to characterize as such. To see this, assume 2-dimensional points whose values in each dimension are real numbers in $[0, 10)$. Assume also that we use 10 buckets and equal partitions, so that values in $[i, i+1)$ are mapped to bucket i, $0 \leq i \leq 9$. Now, take the two points (7.56, 5.30) and (7.23, 6.71). None of them dominates the other, thus

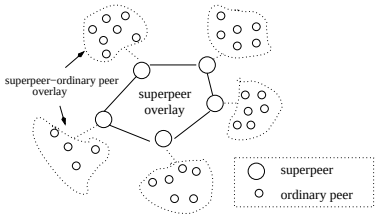

Figure 2: Our generic superpeer architecture

both belong to the skyline. However, using buckets, the two points are represented as (7, 5) and (7, 6) and the second point dominates the first. Thus, the first point would not belong to the skyline.

Note that the points that are not included in the skyline are points whose value in some dimension is only *slightly* better than the value of some other point, so that they both fall into the same bucket. Clearly, the difference in values that is necessary so that a skyline point is not missed depends on the number of buckets.

We can avoid false negatives, if we maintain extended skyline points, that is, if we maintain all points that are either better *or equal* in at least one dimension. This method introduces *false positives*, since there are points that belong to the extended skyline but do not belong to the sklyline.

For improving the accuracy of the representation, we could also use non-equal partitions of the domain. In particular, we could assign more buckets to large values in the domain and less buckets to small values. For example, assume that we use 16 buckets to represent the domain $(0, 10)$. We could allocate 8 buckets to the range $[8, 10)$ and the remaining 8 buckets to the range $(0, 8)$. We call this strategy *adaptive bucket allocation*.

3. BITPEER

In this section, we describe the deployment of our bitmap approach in a distributed setting.

3.1 System Model

We assume a peer-to-peer (p2p) system with N nodes or peers. Each peer P_i maintains a set of data points S_i in space D. Let $S = \cup_i S_i$. We are interested in computing skyline queries on S in any subspace U, $U \subseteq D$. Such queries may be posed by any of the N peers.

In a p2p system, each peer connects to (knows about) a small number of other peers. Thus, an overlay network among peers is formed on top of the physical one. In this paper, we assume a superpeer overlay architecture. In such architectures, a small number of peers, called *superpeers*, are assigned special roles. In general, the peers that are selected to act as superpeers are nodes with good stability properties and sufficient computational and communication resources. We call the peers that do not act as superpeers *ordinary peers*. Each ordinary peers is assigned to one or more superpeer.

In a superpeer architecture, the overlay network can be though of as consisting of two sub-overlay networks: the superpeer overlay and the superpeer-ordinary peers overlay (Fig. 2). The superpeer overlay connects the superpeers with each other. The superpeer-ordinary peers overlay refers to the connections between a superpeer and the ordinary peers assigned to it.

There has been a lot of research work both on building appropriate overlays and on selecting superpeers (for example, [14, 7]). In this paper, we make no specific assumptions with regards to these overlays. We just assume that the superpeer overlay network is connected and that, for simplicity, each ordinary peer is assigned to a single superpeer.

3.2 Subspace Skyline Computation

3.2.1 Bitmap Maintenance

Each superpeer computes and maintains the extended skyline of all ordinary peers that are assigned to it. A bitmap representation is used for representing the extended skyline points.

Upon joining the network, each ordinary peer computes the extended skyline of its local dataset and sends it to its superpeer using the bitmap representation. The superpeer integrates these points to its extended skyline using the bitmap algorithm. The addition of a data point in the local set of an ordinary peer needs to be communicated to its superpeer only if this point belongs to the local extended skyline of the peer. In this case, just this point is transferred to the superpeer.

Deletion of data points is more expensive. Again, deletion of points that do not affect the extended skyline are not communicated to the superpeer. However, deleting a data point that belongs to the local extended skyline of an ordinary peer requires the recomputation of the extended skyline, since this point may have dominated and thus excluded from the skyline other points. To avoid overwhelming the network with updates, updates are performed in batches. An ordinary peer communicates its updates to its superpeer only when $u\%$ of its dataset has been updated.

3.2.2 Skyline Processing

Assume that a peer P_i poses a subspace skyline query. The query is forwarded to the superpeer assigned to P_i. The computation of the query is then the responsibility of the superpeers. The exact routing of the query among the superpeers depends on the type of the superpeer overlay. The minimum requirement is that the routing mechanism visits all superpeers. For improving performance, ideally, each superpeer should be contacted only once. When the underlying superpeer overlay is not acyclic, a practical way of reducing duplicate messages is by assigning a unique *id* to each query. Superpeers that receive a query with the same *id* do not process the query any further.

In general, each superpeer P_2 that receives a subspace skyline query q from a superpeer P_1 forwards the query to its overlay neighbors. Thus, a routing tree is formed rooted at the superpeer that initiated the query. In turn, each superpeer forwards the query to its own neighbors which forward it to their own neighbors and so on. Leaf superpeers compute the extended subspace skyline query using the extended skyline locally stored at them. Then, they forward the resulting data points in bitmap format to the superpeer from which they received the query and so on. A superpeer that receives the results of a query from its children integrates these results to compute the extended subspace skyline query. In particular, given two sets *ext-SKY*(S_1) and *ext-SKY*(S_2) of extended subspace skyline points on datasets S_1 and S_2, the superpeer computes the

extended skyline of their union, $ext\text{-}SKY(S_1 \cup S_2)$. It also integrates these results with its own extended skyline points. To this end, the optimizations previously described are applied. For example, an appropriate point p is selected from $ext\text{-}SKY(S_1 \cup S_2)$ and used to eliminate as many points as possible by computing the $DNSK$ points in the local extended skyline.

4. CACHING

Superpeers cache intermediate results of the skyline computation. The main reason for caching is reducing the communication and computational cost by re-using results of previous queries.

The cache at each superpeer includes results of previously computed skyline queries. Each cache entry includes:

- the *id* of the query that includes its dimensions,
- the query result in Bitmap representation, and
- the list of superpeers that participated in the computation of the cache entry.

Note that for the computation of a subspace query, at all intermediate steps, we compute the extended skyline to avoid false negatives. Consequently, cached results refer to extended skyline queries.

Each superpeer that receives the (intermediate) results of a subspace skyline query caches them in its local cache before forwarding them any further. When a superpeer receives a query, first, it checks whether the same query (i.e., a skyline in the same subspace) has been previously cached. If so, the query is not forwarded to any of the superpeers in the list associated with the entry. Else, query processing proceeds as usual. When the query result is fully computed, the cache content is refreshed to indicate the new list of superpeers that participated in its computation.

When the cache of a superpeer becomes full, an LRU policy is used to replace the least recently used skyline. We also associate an expiration time with each cache entry. When this time expires, the associated entry is considered obsolete. Entries whose expiration times have expired are replaced first.

Besides using the cached results to answer skyline queries in exactly the same subspace, we use Properties 1 and 2 that relate the skylines of different subspaces to also answer similar queries. We also use the following property:

PROPERTY 3. *Assume a set S of data points in space D. Let U, $V \subseteq D$ and $V \subseteq U$, it holds* ext-$SKY_V(S) \subseteq$ ext-$SKY_U(S)$.

It is easy to show that the above property holds. Let $p \in ext\text{-}SKY_V(S)$ and $p \notin ext\text{-}SKY_U(S)$. This means that there is a point q which ext-dominates p in subspace U. That is, at all dimensions in U, q has values larger than the corresponding values of p. But, since $V \subseteq U$, q has values larger than the corresponding values of p in V as well, that is q ext-dominates p in V which contradicts our assumption.

Based on Properties 1, 2 and 3, we make the following observations: (a) To compute the skyline of a subspace U, it suffices to use the extended-skyline of any subspace V such that $V \supseteq U$. (b) The extended skyline of a subspace V is a subset of the extended skyline of any subspace V such that $V \supseteq U$.

We use observation (a) to compute a subspace skyline on U using any available cached extended skyline query on any V, such that $V \supseteq U$. In particular, when a superpeer receives a query at subspace U, it checks its cache for any query at any subspace $V \supseteq U$. If there exists more than one such query, then the one with the smallest set of dimensions is used for computing the skyline in U.

We use observation (b) to compute fast some of the points in a subspace U using any cached extended skyline query on any V, such that $V \subseteq U$. In particular, it holds than any point that belongs to the extended skyline of V also belongs to the extended skyline of U. Thus, we can just output the points in $ext\text{-}SKY_V$ without any additional computation. However, such results are incomplete. In general, $ext\text{-}SKY_V(S)$ is not equal to $ext\text{-}SKY_{V_1}(S) \cup ext\text{-}SKY_{V_2}(S)$, even for $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. For instance, take the points $p_1 = (1, 1, 2)$, $p_2 = (2, 2, 1)$, $p_3 = (4, 1, 1)$ and $p_4 = (2, 3, 2)$ of the example in Fig. 1. $ext\text{-}SKY_{\{d_1,d_2\}} = \{p_2, p_3, p_4\}$, $ext\text{-}SKY_{\{d_1\}} = \{p_3\}$ and $ext\text{-}SKY_{\{d_2\}} = \{p_4\}$.

If there are cached results of more than one subset of U, using all of them would provide us with as much results as possible. However, this could lead to duplicates in the answer. Take again the four points of our example. Note that $ext\text{-}SKY_{\{d_3\}} = \{p_1, p_4\}$ and recall that $ext\text{-}SKY_{\{d_1,d_2\}} = \{p_2, p_3, p_4\}$. Thus, if we use these two sets to compute the $ext\text{-}SKY_{\{d_1,d_2,d_3\}}$, point p_4 would be output twice. To decrease the number of duplicates, we use the following simple heuristic. We choose subsets such that: (a) each one has a large number of dimensions and (b) their overlap with each other is also large.

5. CONTINUOUS SUBSPACE SKYLINE QUERIES

An important type of skyline queries are monitoring queries, where users are interested in being notified of any new points in the skyline. In such cases, users pose continuous skyline queries.

5.1 Overall Architecture

As opposed to simple queries that are executed only once, a *continuous* query is executed repeatedly [6, 4]. Based on the condition that determines their re-evaluation, we distinguish continuous queries as (a) periodic and (b) update-based ones. In *periodic continuous* queries, the user specifies the frequency of execution of the query. In *update-based continuous* queries, a query is re-evaluated when a specific number of updates has occurred on the datasets. We consider both types of queries. Periodic queries include a time period that specifies the frequency of their re-evaluation, while update-based ones include a value that specifies the number of updates that triggers their re-evaluation.

We assume that each continuous query is assigned to one superpeer that becomes responsible for its execution. Each superpeer maintains the extended skylines of all continuous queries assigned to it using a bitmap representation. The superpeer responsible for a continuous query cooperates for its execution with the other superpeers. All superpeers are aware of the continuous queries that are currently in the system and the superpeers that are responsible for each of them. They communicate any updates in the dataset of their ordinary peers to the superpeer responsible for the query affected by these updates.

Next, we discuss how to assign continuous queries to superpeers for sharing computation among them by creating groups of queries. Grouping also makes it possible that superpeers need not be aware of all continuous queries but just one query per group.

5.2 Grouping of Continuous Skyline Queries

Assume a set C of continuous subspace skyline queries and a set SP of superpeers. One could assign queries to superpeers randomly. However, this could lead to repetition of computation, since queries assigned to different superpeers may have results in common.

5.2.1 Similarity-based Grouping

Our goal is to group subspace queries so that, by taking advantage of the containment properties of their result sets, we avoid re-computing the same results many times. In particular, we want to create groups of queries so that the queries in each group are similar to each other and we can compute all queries in a group by just computing an appropriate subset of them.

Let $C_i \subseteq C$ be such a group and let $q_1, q_2, \ldots, q_k$ in subspaces with dimensions $c_1, c_2, \ldots, c_k$ respectively, be the queries in C_i. Our first objective is for the queries in the group to satisfy the following condition:

Condition 1: $c_1 \subset c_2 \subset \ldots \subset c_k$.

If it is not possible to create such groups, we seek for groups whose queries satisfy the following:

Condition 2: $\exists\ c_i$, $1 \leq i \leq k$, such that $c_j \subset c_i$, for all j, $1 \leq j \leq k$, $j \neq i$.

For each group C_i, we choose one query as its representative. The query chosen is the one whose dimensions are a superset of the dimensions of a all queries in the group. Note that for the groups that satisfy Condition 1 or Condition 2, the representative query belongs to the group. It is query c_k in the first case and query c_i in the second one. If groups do not satisfy any of these properties, the representative query may not belong to C_i.

From the properties of the extended skyline, it holds that each query in a group can be computed by just using the extended skyline of the representative of the group. Thus, the responsible superpeer maintains just the extended skyline of the representative query for the group. It also gets notified only for updates concerning this representative. All other queries are computed using the extended skyline of the representative.

In particular for groups satisfying Condition 1, the computation of their queries can be performed very efficiently. The computation starts with the query with the largest number of dimensions among the queries in the group and proceeds with the query with the second largest number and so on. Each query in this sequence is not computed using the extended skyline of the representative, but, instead using the results of the query previously computed.

For creating groups that satisfy Conditions 1 or 2, we use the following heuristics:

1. We partition the set C of queries based on the number of their dimensions. Let C_m be the set of all queries with m dimensions, $1 \leq m \leq d$. Note that none of the C_ms satisfies any of the two conditions.
2. Among the C_ms, we choose the set C_k that has the largest number of elements (i.e., queries). We create $|C_k|$ groups, one for each query.
3. We assign each query in C to one of the $|C_k|$ groups, so that either Condition 1 or Condition 2 holds. If this is not possible, we create a new group.

Let M be the number of groups thus created. Each group is assigned to one superpeer. If the number of groups is larger than the number of superpeers ($M > SP$), we may assign more than one group to some superpeers. If we want to create less than M groups, we merge some of the created groups. In particular, we choose to merge two groups C_i and C_j if the queries in C_i have a large number of overlapping dimensions with the queries in C_j.

5.2.2 Grouping for Periodic Queries

We focus now on periodic continuous queries. In this case, each query q_i is re-evaluated every t_i time units. The grouping described in the previous section is applicable to this case as well. It suffices to assign to the representative query of each group a time period equal to the smallest time period of all queries in its group. However, this may lead to unnecessary computations, especially when the time frequencies of the queries in the group vary widely.

In this case, it is possible to group queries based on their t_i values. Let $C_i \subseteq C$ be such a group and let $q_1, q_2, \ldots, q_k$ with time periods $t_1, t_2, \ldots, t_k$ respectively, be the queries in C_i. Our first objective is for the time intervals of all queries in the group to satisfy the following condition:

Condition 3: $t_1 = t_2 = \ldots = t_k$.

If this is not possible, then we look for groups such that:

Condition 4: There are positive integers β_i, $1 \leq i \leq k$, such that $\beta_1\ t_1 = \beta_2\ t_2 = \ldots = \beta_k\ t_k$.

Again, we consider as representative for each group a query in the subspace having as dimensions the union of the dimensions of all queries in the group. The period t_i of this query is the smallest period of the queries in the group.

It is also possible to combine the two types of grouping, that is Condition 1 and 2 with Conditions 3 and 4. One way is by seeking for groups that satisfy both types of conditions. Alternatively, we could look for groups that satisfy one of the two types and then partition the resulting groups so that they also satisfy the other type.

6. EXPERIMENTAL EVALUATION

6.1 Data Distribution

We study datasets following the benchmark databases used in [2]. In particular, points are generated using one of the following distributions:

- *Correlated*: In a correlated dataset, points which are good in one dimension tend to be good in other dimensions as well. As a result, a fairly small number of data points dominates many other points, thus the size of the skyline is rather small.
- *Anti-correlated*: In an anti-correlated dataset, points which are good in one dimension are bad in the other dimensions. As a result, the skyline is typically large.

Table 1: Number of skyline and extended skyline points using a bitmap representation with and without buckets

Number of Buckets	Number of Skyline Points			Number of Extended Skyline Points		
	Independent	Correlated	Anti−Correlated	Independent	Correlated	Anti−Correlated
8	164	1195	715	2326	1388	9326
10	101	935	899	1843	1129	9115
16	21	279	808	925	651	6527
32	5	219	174	576	428	2314
64	3	59	89	277	201	767
100	3	22	58	175	132	399
1000 (no buckets)	5	3	90	25	20	145

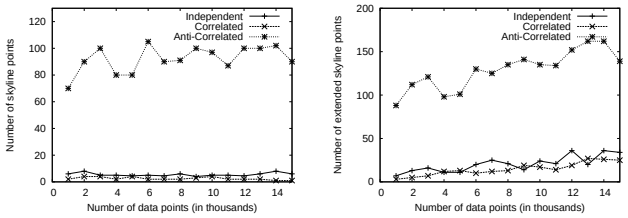

Figure 3: Number of (left) skyline and (right) extended skyline points

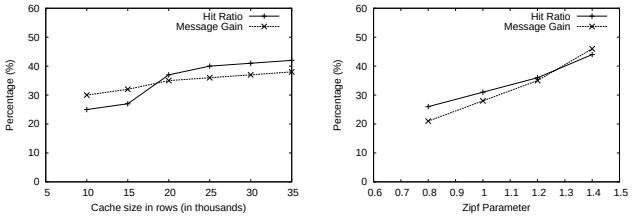

Figure 4: Hit ratio and communication savings with (left) cache size, (right) query locality (zipf parameter α)

- *Independent*: In an independent dataset, points are generated using a uniform distribution. The size of the skyline is between that of the correlated and the anti-correlated datasets.

The points in the dataset are distributed among the peers. We performed three sets of experiments. In the first set, we study the size of the extended skyline and thus the overhead of pre-computing extended skylines versus pre-computing simple skylines. In the second set, we focus on the benefits of caching, while in the third set we consider grouping for continuous queries.

6.2 Extended and Bucket Skyline

In this set of experiments, we consider the increase of the size of the skyline and the extended skyline with the number of points in the dataset. The results for two-dimensional points are depicted in Fig 3. As expected, the number of both the skyline points and the extended skyline points for the anti-correlated data set is much larger than that for the other two distributions. The relative increase in the size of the extended skyline over the simple skyline is the smallest (around 35%) for the anti-correlated case.

In Table 1, we show how using buckets for the Bitmap representation of data points affects the skyline and the extended skyline computation. The domain of each dimension has 1000 distinct values. We have a total of 10000 points. In this experiment, we used 8, 10, 16, 32, 64 and 100 buckets. We also show the corresponding sizes of the skylines, when no buckets are used. Observe that when we use buckets to compute the skyline points, the points may be less than when we do not use buckets. This is because in this case, we may have false negatives. That is, the algorithm may fail to identify some skyline points. Using extended skyline points alleviates this problem. It also allows us to compute any subspace skyline query. However, the size of the extended skyline (both with and without buckets) is much larger. In the results reported in Table 1, we used buckets having equal sizes.

Using the adaptive bucket approach works well for the independent and the correlated approach reducing the number of points. It does not work well for the anti-correlated distribution, since points that are good at one dimension are less good at the other, thus, there are no important values to which to assign more buckets.

6.3 Caching

In this set of experiments, we study caching in terms of (a) cache hit ratio: the number of times a query uses the results in the cache and (b) message gain: the reduction of messages due to caching. We report results for 6-dimensional data points generated using the independent distribution; the results for the other two distributions are analogous. We start with an empty (cold) cache. We have 1000 peers of which 100 act as superpeers. We generate the sub-space skyline queries using a zipf distribution with $\alpha = 1.2$. As shown in Fig. 4(left), even a relatively small cache results in important savings. The cache benefits increase with locality, as shown in Fig. 4(right) for a cache with 20000 points.

6.4 Continuous Queries

In this experiment, we consider continuous processing of subspace skyline queries. The performance of similarity-based grouping depends on the overlap among the dimensions of the sub-space queries. We consider 20 continuous subspace queries. We consider creating 4, 8, 12, 16 and 20 groups using the conditions in Section 5.2. Note that 20

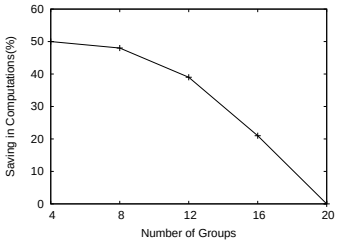

Figure 5: Benefits of similarity-grouping with overlap

groups means that each group consists of a single query, that is, we have no grouping. As shown in Fig. 5, grouping results in large savings in computation.

7. RELATED WORK

There is a variety of methods for efficient skyline computation including block-nested loop, sorting and indexing [2, 10]. In this paper, we adapt a bitmap approach that provides a compact representation of the data set that is very important in terms of communication efficiency.

Subspace skyline queries have also attracted attention recently mainly for centralized environments. The authors in [9] introduce *skyline groups* which are groups of objects that are coincidental in the skylines of some subspaces. They also identify the *decisive subspaces* that qualify skyline groups in the subspace skylines. They develop *Skyey* algorithm for computing the skyline groups and the subspace skylines. The authors in [15] consider the efficient computation of the skylines of all possible subspaces, called the *SKYCUBE*. Their focus is on sharing strategies by identifying the computation dependencies among multiple related subspace skyline queries.

In [1], the authors consider distributed skyline computation over a dataset that is vertically distributed, that is, partitioned based on dimensions. In this paper, we focus on a horizontal distribution, which is very common in p2p systems, where each peer maintains its own dataset. The work in [5] considers skyline computation in unstructured p2p systems. In contradistinction to our superpeer approach where superpeers coordinate with each other to cover the whole dataset, in unstructured p2p systems, each peer knows only its neighboring peers. Thus, the focus is on providing probabilistic guarantees for the computation of the skyline. The authors of [13] consider skyline queries in a structured p2p overlay, where items are mapped to peers based on their values. The assignment is based on recursive region partitioning and encoding. Thus, efficient skyline query processing can be achieved, however, data needs to be moved. Similarly, in [12], the authors adapt a structured approach and use BATON, a distributed index structure, to compute skylines. The most similar approach to ours is perhaps *SKYPEER* [11]. The authors consider subspace skyline processing in a superpeer based system using extended skylines. Here, we propose a method based on a bitmap representation for realizing such computations. We also consider caching and continuous queries.

Finally, there has been a lot of work on publish/subscribe systems, recently also in the context of content-based overlays (e.g., [8, 3]), where users subscribe their interests on specific pieces of data and get notified when related items are published. A nice extension of our approach would be to integrate our continuous skyline queries as a special type of subscription queries.

8. SUMMARY

In this paper, we have presented an approach for computing skyline queries in a distributed setting that is based on a bitmap representation. Bitmap-based representations are appropriate for distributed computing, because they support compact representations of the datasets that can be efficiently transmitted over the network. They are also amenable to encodings and compression. We have presented algorithms and heuristics for efficient bitmap-based computation of extended skyline queries. We have also proposed a caching scheme and a grouping method for continuous queries.

9. REFERENCES

[1] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, 2004.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[3] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.

[4] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, 2000.

[5] K. Hose, C. Lemke, and K.-U. Sattler. Processing relaxed skylines in pdms using distributed data summaries. In *CIKM*, 2006.

[6] L. Liu, C. Pu, R. S. Barga, and T. Zhou. Differential evaluation of continual queries. In *ICDCS*, 1996.

[7] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. T. Schlosser, I. Brunkhorst, and A. Löser. Super-peer-based routing and clustering strategies for rdf-based peer-to-peer networks. In *WWW*, 2003.

[8] A. M. Ouksel, O. Jurca, I. Podnar, and K. Aberer. Efficient probabilistic subsumption checking for content-based publish/subscribe systems. In *Middleware*, 2006.

[9] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, 2005.

[10] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, 2001.

[11] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis. Skypeer: Efficient subspace skyline computation over distributed data. In *ICDE*, 2007.

[12] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu. Efficient skyline query processing on peer-to-peer networks. In *ICDE*, 2007.

[13] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*, 2006.

[14] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *ICDE*, 2003.

[15] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, 2005.