International Journal of Data Science and Analytics (2019) 7:17–34
https://doi.org/10.1007/s41060-018-0098-x

REGULAR PAPER

BJR-tree: fast skyline computation algorithm using dominance relation-based tree structure

Kenichi Koizumi[1] · **Peter Eades**[2] · **Kei Hiraki**[3] · **Mary Inaba**[1]

Received: 14 November 2017 / Accepted: 19 January 2018 / Published online: 31 January 2018

Abstract
High-throughput label-free single-cell screening technology has been studied for the noninvasive analysis of various kinds of cells. Selecting the prominent cells with extreme features from a large number of cells is an important and interesting problem, which we call the serendipitous searching problem (SSP). In the SSP, it is important to find entries located near the rind of the population in a multi-dimensional feature space. We tackle the SSP as a continuous skyline computation. Originally, the skyline computation was designed to extract interesting entries from a database with multi-attributes. The skyline points are continuously updated as the existing entries disappear and new entries arrive. In this paper, we propose a balanced jointed rooted tree (BJR-tree) algorithm and a non-dominated relation cache (ND-cache) for continuous skyline computation. The BJR-tree expresses the dominance relation as an arc and stores the "dominated" relations. The ND-cache complements the BJR-tree by reducing the recalculation of the dominance relations. The execution times of the BJR-tree and existing continuous skyline computation algorithms are compared on randomly constructed synthetic datasets with multiple temporal and spatial features. The BJR-tree is then evaluated on actually measured information of blood cells. On the two- and eight-dimensional synthetic datasets, the BJR-tree computed the continuous skylines approximately 3 and 70 times faster than LookOut, respectively. On real-world datasets, BJR-tree was approximately 2.4–3.2 times faster than LookOut.

Keywords Algorithm · Streaming application · Continuous skyline computation

1 Introduction

1.1 Skyline computation

Skyline computation algorithms [9] are used for extracting the interesting entries from a database of multi-attribute entries. Skyline computation plays an important role in data-driven artificial intelligence, which extracts the extraordinary entries from data without requiring a model. This multi-objective optimization problem seeks a Pareto (maxima) set of vectors [34], and various efficient algorithms (e.g., [35]) are studied. The computation can be stated as a geometric problem. Given two d-dimensional vectors $\boldsymbol{v} = (v_0, v_1, \ldots, v_{d-1})$ and $\boldsymbol{u} = (u_0, u_1, \ldots, u_{d-1})$, we define the *dominance relation* as follows: $\boldsymbol{v}$ *dominates* $\boldsymbol{u}$ if and only if $(\forall k \in \{0, 1, \ldots, d-1\}\ v_k \leq u_k) \bigwedge (\exists k \in \{0, 1, \ldots, d-1\}\ v_k < u_k)$. Given a set V of n points in the closed positive orthant of a d-dimensional space, $\boldsymbol{v} \in V$ is defined as a *skyline point* of V if there exists no $\boldsymbol{w} \in V$ such that $\boldsymbol{w}$ dominates $\boldsymbol{v}$. The set of all skyline points of V is called the *skyline* of V. The left and right panels of Fig. 1 show

This paper is an extension version of the DSAA'2017 Research Track paper titled "BJR-tree: Fast Skyline Computation Algorithm for Serendipitous Searching Problems".

✉ Kenichi Koizumi
koiken@is.s.u-tokyo.ac.jp

Peter Eades
peter.eades@sydney.edu.au

Kei Hiraki
hiraki@is.s.u-tokyo.ac.jp

Mary Inaba
mary@is.s.u-tokyo.ac.jp

[1] Department of Creative Informatics, The University of Tokyo, Tokyo, Japan

[2] School of Information Technologies, University of Sydney, Sydney, Australia

[3] Department of Chemistry, The University of Tokyo, Tokyo, Japan

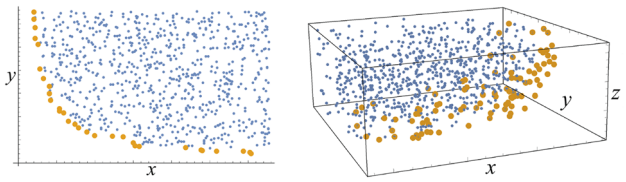

Fig. 1 Examples of skylines in multi-dimensional spaces (left: two dimensions and right: three dimensions). Yellow and blue points denote the skyline and non-skyline points, respectively

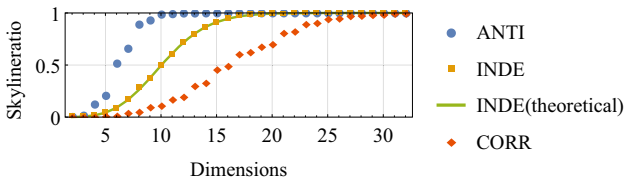

Fig. 2 Skyline ratio versus number of dimensions for 10,000 randomly generated entries with three types of spatial distributions (anti-correlated, independent, and correlated)

examples of skylines in two- and three-dimensional spaces, respectively. As shown in Fig. 2, the skyline cannot extract entries effectively from a high-dimensional space because the skyline ratio (ratio of skyline entries to whole entries) asymptotically approaches 1. The average skyline ratio r of n d-dimensional entries is obtained as follows [11]:

$$r = \frac{1}{n}\left(\sum_{k=1}^{n}(-1)^{k+1}\frac{\binom{n}{k}}{k^{d-1}}\right) \tag{1}$$

We experimentally confirmed that when r is fixed, the number of dimensions d is proportional to $\log n$. Therefore, we target mid-range-dimension even for larger n.

The task of computing the skyline of a *dynamic* set of points is known as the *continuous skyline computation* [40]. Real databases are updated frequently in streaming or other suitable environments. Streaming applications play a significant role in diverse fields such as fraud detection in financial trading, intrusion detection in computer networks, data processing, scheduling, and traffic control in sensor network applications [44]. The injection and ejection times of an entry into/from the database are called the *activation* and *deactivation* times of the entry, respectively. Let $act(\boldsymbol{v})$ and $deact(\boldsymbol{v})$ be the activation and deactivation times of an entry $\boldsymbol{v}$. At time t, define S_t as the skyline of $V_t = \{\boldsymbol{v} \in V \mid act(\boldsymbol{v}) \leq t < deact(\boldsymbol{v})\}$. In a continuous skyline computation, S_t is updated from the points that are activated and deactivated at time t.

A continuous skyline computation is useful for filtering out non-skyline points in preprocessing for screening a large amount of data in real time. Our objective is to optimize

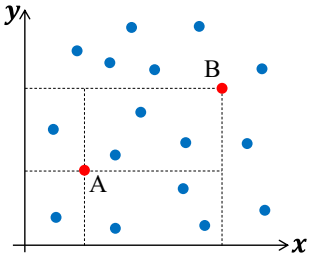

Fig. 3 Points with high and low skyline potentials (Points A and B, respectively)

the overall maintenance time of the continuous skyline computation. The continuous skyline computation is a difficult problem because the activation or deactivation of just one point can demote many existing skyline points from the skyline or promote many existing points to the skyline, which changes a large part of the skyline. Many previous studies have proposed data structures for managing point sets and fast algorithms for computing the skyline. These methods use spatial-indexing trees, such as B-trees, R-trees, R*-trees [9,31,42], or quadtrees [40].

1.2 New tree structure for skyline computation

We can roughly measure the likelihood that a point will become a skyline point in the near future using the concept of *skyline potential*, define as follows. We employ the skyline potential of point $\boldsymbol{v}_A$ in the d-dimensional space defined by

$$1 - \sqrt[d]{\frac{|\{\boldsymbol{v} \in V \mid \boldsymbol{v} \text{ dominates } \boldsymbol{v}_A\}|}{n}} \tag{2}$$

A skyline potential of a skyline point is 1. Assuming that points are uniformly distributed, the number of points is proportional to the hypervolume. Therefore, we use the d-th root of the dominant point ratio. A point that is dominated by many points has a low potential to become a skyline point. As the points in Fig. 3 are activated and deactivated, the skyline potential of point A exceeds that of point B. Exploiting this difference in skyline potential, we can reduce the number of comparison operations in the skyline update.

Existing algorithms can compute the skyline of low dimensional of data (e.g., $d \lesssim 5$) at reasonably fast speed. However, because points are stored in the tree without considering their potential to become a skyline point, these algorithms perform redundant computations. Additionally, the number of spatial partitions in a spatial-indexing tree exponentially increases with number of dimensions. Therefore, the points become sparsely distributed in the partitioned space, and the traversal performance of the tree decreases. It is inefficient to store every dominance relation between pairs of active points in the continuous skyline computation

in a simple $O(N^2)$-sized table. As the dominance relation is transitive, there are numerous dominance relations. However, many of these dominance relations (such as those between points with low potential of joining the skyline) are unlikely to be used in the skyline computation, so their calculations can be delayed. Furthermore, the number of recalculations can be reduced by storing the results of the previous activation and deactivation operations. We propose a *balanced jointed rooted tree (BJR-tree)* for continuous skyline computation. Each vertex and arc in the BJR-tree represents a point and a dominance relation, respectively, but not every dominance relation is stored. As the dominance relation is transitive, a directed graph G that exactly expresses a set of points V is uniquely determined. This graph, called the *complete dominance graph*, is a transitive closure with $\Omega(n^2)$ arcs, whereas the BJR-tree contains $O(n)$ arcs. Our concept excludes arcs that are unnecessary for computing the skyline and preserves arcs that may be utilized in the future. The BJR-tree is a rooted spanning subtree of the complete dominance graph.

The higher computational speed of the BJR-tree algorithm over existing algorithms is attributed to two features: (1) appropriate hierarchical expression and (2) dimensionality independence.

Regarding (1), the depth of nodes in the BJR-tree reflects the potential of the entries to join the skyline, namely the *skyline potential*. A point with higher potential is closer to the root than a point with lower potential. Existing spatial-indexing methods (such as B-tree, R-tree, and quadtree), which express the clustering and proximity relationships, are unsuitable for skyline computation because the non-uniformity of a spatial distribution is not directly related to the dominance relationships. Regarding (2), BJR-tree algorithms are dimensionality-independent because they project a multi-dimensional space onto a simple graph using the dominance relations alone. In other words, the number of comparisons is independent of the dimensionality, so the skyline computation can be rapidly and continuously performed in any number of dimensions. Furthermore, the BJR-tree is not restricted to a set of points with a dominance relationship, but applies to any partial order. The original JR-tree (on which the BJR-tree is based) was implemented in the winning entry of the 2015 MEMOCODE Design Contest [30,39]. This paper is an extension version of [29]. The BJR-tree is detailed in Sect. 3, and its complexities are discussed in Sect. 5.

We then propose a non-dominated relation cache (ND-cache), a data structure that accelerates the continuous skyline computation. This $O(n)$-sized structure essentially caches the non-dominated relationships (in contrast to the BJR-tree, which stores dominance relationships). The BJR-tree and ND-cache play complementary roles.

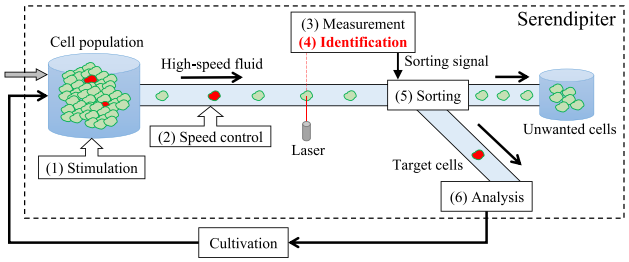

Fig. 4 Flow diagram of Serendipiter

1.3 Serendipitous searching problem

Many classification methods for spatial data processing in super-high-dimensional spaces have been proposed in recent years. Clustering methods divide vectors into several groups, and outlier detection methods identify distinct entries such as noise. The *serendipitous searching problem (SSP)* is a new cognitive problem that detects "boundary" entries, i.e., entries that are located geometrically near the "rind" (or envelope) of a population. The population may comprise multiple clusters or include outliers. The typical characteristics of the population are regarded as central entries (e.g., entries around the mean and median). Meanwhile, the extreme characteristics of the population are regarded as boundary entries. Very rare and valuable boundary entries can be encountered serendipitously.

We solve the SSP as a skyline computation problem, regarding the set of boundary entries as a set of skyline points. As the cultivation proceeds, the features of the boundary set gradually shift. Hence, the boundary group is maintained by an online algorithm that adds newly processed cells and removes earlier cells.

Our SSP application *Serendipiter* [22,24,41] is a fast cell sorter that discovers very rare cells with atypical ability from an enormous number of cells. Serendipiter includes six technologies at the single-cell level with the following functionalities: (1) cell stimulation, (2) speed control of the cells, (3) high-resolution cell measurements using multiple sensors, (4) identification of cells from multi-dimensional vectors of cell measurement information, (5) sorting of cells into wanted and unwanted groups, and (6) analysis of the wanted cells. A block diagram of Serendipiter is shown in Fig. 4. The cells are identified from cell measurement information obtained by multiple sensor technologies, such as optical imaging and spectroscopy. The measurement and identification latencies must be below 10 ms. Existing cell sorters such as [3] process cells at sufficient speed, and microscopes enable the accurate analysis of cells. However, because these methods cannot simultaneously realize fast and accurate analysis, we have developed Serendipiter. The discovery of very rare cells (constituting one per trillion cells) in a realistic time is

expected for efficient biofuel production by *Euglena* spp. and high-precision blood testing. Serendipiter, which combines an optical time-stretch quantitative phase microscope with a hydrodynamic-focusing microfluidic chip, accurately analyzes single cells and classifies 10,000 cells each second.

In biofuel production by *Euglena* spp., we require cells with a superior fat-producing ability (Super *Euglena*). However, because the measured static information relies on the lifecycle and photonic synthesis phase of the cells, it cannot directly determine the fat production potential of *Euglena* cells. Various features of *Euglena* cells can be generated by stimulating the cells and promoting mutation. The Serendipiter system selects and cultivates *Euglena* cells with extreme features. After several stimulation–cultivation cycles, we obtain a population of *Euglena* cells that are rarely found in ordinary cell populations. The efficiency of our proposed algorithms is verified on Serendipiter.

To determine the usefulness of the skyline cells extracted by BJR-tree, we must biochemically analyze the cells after repetitive cultivation and extraction. This paper confirms that by appropriately selecting the features, we can stably isolate rare skyline cells. We also confirm the faster execution of BJR-tree than the existing algorithms, regardless of feature selection.

2 Related work

In a pioneering study, Kung et al. [34] extracted the maxima among a set of vectors. They also showed that for n vectors in d-dimensional space, the time complexity of this problem is $O(n \log_2 n)$ for $d = 2, 3$, and it is bounded between $\lceil \log_2 n! \rceil$ and $O(n \log_2^{d-2} n)$ for $d \geq 4$. The same analysis has been applied to database applications, e.g., the skyline operator of Börzsönyi [9].

In static skyline computation, the input points are loaded and stored in data structures, which are accessed by a skyline computation algorithm. Many algorithms have been developed for static skyline computation; here we mention some of them. The block nested loop (BNL) [9], bitmap, and index [50] algorithms use no special data structures. A variant of the BNL algorithm, called the sort-filter-skyline (SFS) algorithm [16], presorts the points. The linear elimination sort for skyline (LESS) algorithm [20] improves the SFS algorithm. The sort and limit skyline algorithm (SaLSa) [1] presorts points by a monotonic limiting function. The authors of [9] proposed basic divide-and-conquer algorithms using a B-tree [2] or R-tree [23], and those of [31] proposed nearest-neighbor algorithms using an R*-tree [4]. The branch and bound skyline (BBS) algorithm [42] also uses an R*-tree. In the object-based space partitioning (OSP) algorithm [56], the partition trees store only the skyline points. BSkyTree [35] selects effective pivots for the space partitioning, which reduces the number of dominance relation computations. These methods, which construct spatial-indexing tree structures, initially build a tree from the input points and then traverse the tree to compute the skyline. These methods are optimized for static skyline computation and are unsuitable for updating the skyline.

Depending on the format of their temporal information, datasets in continuous skyline computation are classified into three types: count-based [38,51], moving-object-based [25,36,52], and time-based [40]. Count-based datasets include an activation sequence with a fixed number of concurrently activated points. The complexity of the update operation for count-based datasets is discussed in [38]. In moving-object-based datasets, the points move around a multi-dimensional space and the skyline must be tracked throughout the simulation. In [36], the tracking is performed on an R*-tree. Our algorithm targets a time-based dataset with the time of activation (entry into the database) and deactivation (exit from the database).

In continuous skyline computation, the tree-building algorithm is replaced by injection and ejection algorithms. A variant of the bitmap algorithm was proposed in [19]. The LookOut algorithm [40] performs more effectively on a quadtree [18] than an R*-tree. The quadtree reportedly outperforms the R-tree in update-intensive applications [32] and is more effective in point indexing than the R*-tree [28]. The height of an n-leaf quadtree in d-dimensional space is $O(\log_d n)$. Closely placed points are stored in deep positions of the quadtree. Because the quadtree must be recursively traversed in skyline computation, the computational time is difficult to reduce. Recently, many parallelized and hardware-based methods have been proposed for this purpose. For instance, Lazy List and Lock-Free Parallel BNL [47] and APSkyline [37] have been proposed for a multicore processor environment. The hardware-based methods include methods for graphic processing units (GPUs) [6,7,15] and programmable gate arrays (FPGAs) [54,55]. This paper focuses on algorithms for optimization in single-threaded computation.

There are several computational problems that are closely related to the skyline computation. For example, the k-dominant skyline [13] is similar to the skyline, but contains more points under the relaxed dominance relation definition. The nearest-neighbor query [45] and convex hull [21] are well-known extraction problems with multi-dimensional vectors. The SR-tree [27], proposed for nearest-neighbor querying, is a structure that is a combination of an R*-tree and an SS-tree [53]. A method based on X-tree [5], a variant of R-tree, has been proposed for convex hull computation [8].

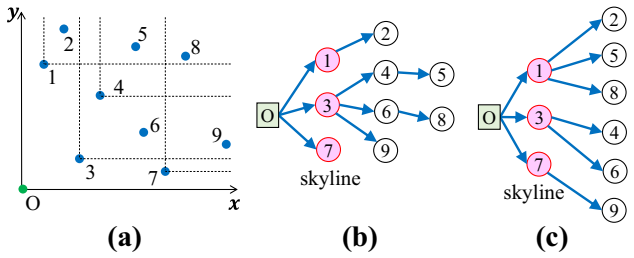

Fig. 5 Examples of BJR-trees **b**, **c** constructed from nine points **a** represented by two-dimensional vectors

3 BJR-tree

3.1 Algorithm description

In the BJR-tree concept, a point that is dominated by many other points is unlikely to become a skyline point in the future, so its precise information need not be maintained. The BJR-tree is a rooted tree that expresses the dominance relations in a given set V, with n points in the closed positive orthant of a d-dimensional space. Each vertex represents a point, and each arc (directed edge) represents a dominance relationship. The vertex set of a BJR-tree is $\{O\} \cup V$, where O is the origin of the d-dimensional space. Note that O dominates all n points in V, and its corresponding vertex is the root of the BJR-tree. A *complete dominance graph* is a directed acyclic graph in which the vertices and arcs correspond to the points $\{O\} \cup V$ and all dominance relations, respectively. A BJR-tree T is a spanning subtree of the complete dominance graph with the following properties: (1) If $\boldsymbol{v}_X$ is a skyline point, then T directly connects O to the corresponding vertex X, and (2) if $\boldsymbol{v}_X$ is not a skyline point, then T directly connects a non-origin point to the corresponding vertex X.

A given set of points in a multi-dimensional space admits multiple BJR-trees. If an arc extends from A to B, then point $\boldsymbol{v}_A$ corresponding to A dominates point $\boldsymbol{v}_B$ corresponding to B. However, even if $\boldsymbol{v}_A$ dominates $\boldsymbol{v}_B$, T does not necessarily connect A to B by one arc. A point belongs to the skyline if and only if the corresponding vertex is a child of O.

Figure 5 shows two examples of BJR-trees constructed from nine points in a two-dimensional space. One of these BJR-trees is sufficient to determine the skyline.

To output the skyline, we need to enumerate the children of O. There is no need to traverse the tree. The BJR-tree offers two advantages in continuous skyline computation: (1) the small size of the BJR-tree and (2) the applicability of BJR-tree to partially ordered datasets as well as multi-dimensional datasets. Regarding (1), the tree contains $O(n)$ edges, whereas the complete dominance graph contains $\Omega(n^2)$ edges. Regarding (2), BJR-tree accesses the values of each dimension when computing the dominance relation, and its concept is applicable to any dataset with a transitive partial order. In contrast, methods based on spatial-indexing trees evaluate the dominance relations between a point and the maximum or minimum corners of the partitioned regions, using the Manhattan distance between the points. In methods that select the dimensions for presorting, the constructed trees are of limited applicability in solving independent multi-objective optimization problems. A point corresponding to a vertex in a deeper layer of the BJR-tree tends to have a low skyline potential, whereas a point corresponding to a vertex near O tends to have a high skyline potential.

Continuous skyline computation activates and deactivates the entries over time. The BJR-tree dynamically injects new vertices and ejects existing vertices. The BJR-tree is built by repeating the injection and ejection operations, which are implemented by Algorithms 1 and 2, respectively.

The BJR-tree maintains the dominated relations discovered in the injection and ejection operations as arcs. This information may be utilized when the skyline potential of the points increases because of the ejection operations. This information is used when related points approach the root under subsequent ejection operations. When executing multiple ejection operations in a single time-step, it is helpful to collectively remove points and inject the children into their parents.

Algorithm 1 Injection (base)

```
1: procedure inject(r: root, v: new vertex)
2: C ← children of r;
3: for all c ∈ C do
4:     if c dominates v then
5:         inject(c, v);
6:         return
7: set v to r's child;
8: for all c ∈ C do
9:     if v dominates c then
10:        move c to v's child;
```

Algorithm 2 Ejection

```
1: procedure eject(v: ejected vertex ≠ O)
2: p ← parent of v;
3: C ← children of v;
4: remove v from p;
5: for all c ∈ C do
6:     inject(p, c);
```

We now show a running example of the injection and ejection operations. Suppose that the BJR-tree in Fig. 5b is constructed from the dataset shown in Fig. 5a and that point 3 is to be deactivated. First, node 3 is removed from the tree. Second, the children of node 3 (nodes 4, 6, and 9) are injected into the root node (the parent of node 3). Node 4 neither dominates nor is dominated by node 1 or 7; consequently, node 4 becomes a child of the root node. Node 6 neither dominates nor is dominated by node 1, 4, or 7, so becomes a child of

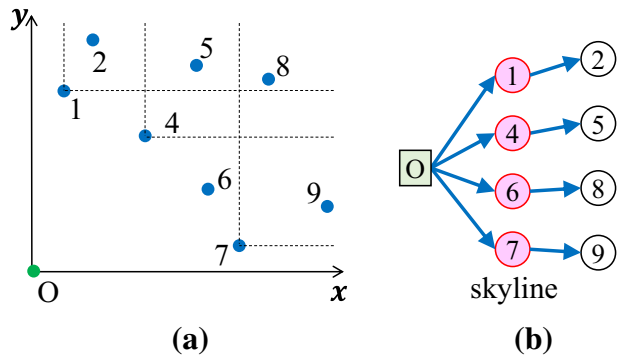

Fig. 6 Points in two-dimensional space (**a**) and a BJR-tree (**b**) derived from the dataset shown in Fig. 5 after ejecting point (node) 3

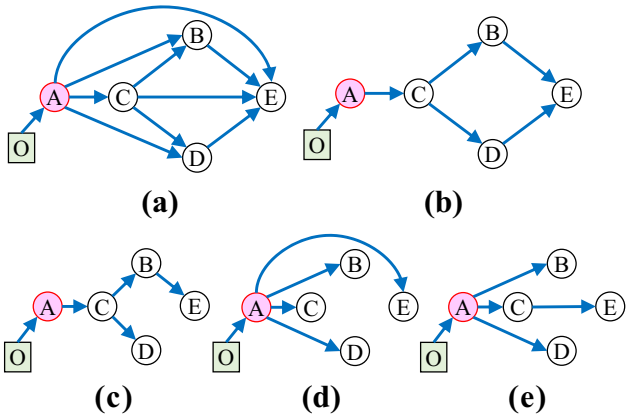

Fig. 7 Examples of rooted trees derived from transitive closure (**a**). The reachabilities of the rooted trees differ from those of the original transitive closure and its transitive reduction (**b**). Tree **c** is generated by Algorithm 1, and trees **d**, **e** are generated by Algorithm 3

the root node. Finally, node 9 is dominated only by node 7, which has no children; therefore, node 9 becomes a child of node 7. Figure 6b shows the new BJR-tree constructed after ejecting node 3 from the BJR-tree in Fig. 5b.

3.2 Lazy strategy

The `inject()` and `eject()` procedures (in Algorithms 1 and 2, respectively) generate one possible BJR-tree representing the current set of points. The generated tree depends on the child-selection order in the first iteration of Algorithm 1. For example, multiple BJR-trees can be generated from the complete dominance graph in Fig. 7a, which is transitively closed. Note that graph (b) of Fig. 7 is the transitive reduction in graph (a) and is not a BJR-tree. Clearly, the rooted tree generated by `inject()` without `eject()` (see Fig. 7c) is a spanning subtree of the transitive reduction.

Algorithm 1 increases the depth of the tree. A shallow vertex in the BJR-tree corresponds to a point with high skyline potential. Such points must be processed immediately. Conversely, points far from the origin (i.e., deeper vertices) can be computed later because their results may not be used until the end of the computation. However, in tree (c) of Fig. 7, the dominance relation between B and E is computed even though both points are more than two hops from A.

To delay such operations, we substitute Algorithm 1 with Algorithm 3. A tree built by Algorithm 3 is displayed in Fig. 7d. This tree is a subtree of the transitive closure and satisfies the conditions of a BJR-tree. Rather than increasing the tree height, Algorithm 3 increases the child number of a parent. By using appropriate thresholds on the dataset, we can balance the number of children and the depth of each vertex. Tree (d) is low height but unbalanced. In contrast, tree (c) has the smallest maximum number of children among the trees in Fig. 7 and is properly balanced, but has a large height. Figure 7e shows another tree built by Algorithm 3, which is a compromise between trees (c) and (d). Building the appropriate type of BJR-tree equates to a lazy evaluation problem. The timing of the lazy evaluation is determined by imposing a limited depth, as shown in Sect. 5.

3.3 Balancing injection

In the previous subsection, we discussed the balance between the tree height and number of children. Here, we balance the numbers of descendants of vertices at the same depth. When multiple vertices dominate an injected vertex, Algorithm 1 selects and traverses from the first-found vertex, which unbalances the BJR-tree. When vertex X is injected to vertex Y and Y has children Z and W who dominate X, either Z or W is chosen, whichever equalizes (as far as possible) the number of descendants of Z and W. This concept, which underlies Algorithm 3, balances the number of descendants of each vertex. The number of descendants of vertex x is denoted by $Desc(x)$. In the ejection process (Algorithm 2), the number of calls of the injection function equals the number of children of the ejected vertex. A single ejection operation invokes multiple injection operations. A BJR-tree is balanced directly in each invoked injection operation. Therefore, the BJR-tree remains balanced through the entire ejection operation, and the ejection process yields a balanced BJR-tree.

4 ND-cache

In the datasets considered here, once a set of skyline points p has been dominated by a set of newly activated points q, some of the p are restored as skyline points by deactivating q. In such datasets, we can reduce the recalculation by saving information about the former dominance relation calculations. Because non-dominated relations (except those of the skyline) are not stored in the BJR-tree, many recalculations are required.

Algorithm 3 Injection (lazy evaluation and tree balancing)

```
1: d: the depth at which a vertex is injected lazily
2: procedure inject(r: root, v: new vertex)
3: if r = O or Depth(r) < d then
4:   C ← children of r;
5:   g ← +∞;
6:   t ← null;
7:   for all c ∈ C do
8:     if c dominates v and Desc(c) < g then
9:       t ← c;
10:      g ← Desc(c);
11:  if t ≠ null then
12:    inject(t, v);
13:    return
14: set v to r's child;
15: if r = O or Depth(r) < d then
16:   for all c ∈ C do
17:     if v dominates c then
18:       move c to v's child;
```

To solve this problem, we propose a non-dominated relation cache (ND-cache) that stores recently computed skylines. The ND-cache comprises an array C of length N (where N is the total number of given points). Each value C_i is updated at time t_1 if and only if point e_i is a skyline point at t_1. At time t_2, if C_a equals C_b, then e_a cannot dominate e_b and e_b cannot dominate e_a. (We call this case a "cache hit.") If C_a and C_b are unequal, the standard comparison is computed.

As an example, Tables 1 and 2 show the history of the skyline and the contents of ND-cache at time T_3 of Table 1, respectively. As e_2 and e_3 are equal in the skyline cache, both points simultaneously belong to the skyline, and neither dominates the other. Table 3 compares the complexities of ND-cache, a non-cached method, and a method storing all former comparison results. The spatial complexity of the method that caches all results is $O(N^2)$, versus $O(N)$ for the ND-cache. The time complexity of a hit is $O(1)$ in a fixed-dimensional space.

We consider the situation in which the skyline S composed of s points is completely dominated by a newly activated point, and one new skyline point is deactivated immediately. The ejection operation invokes $s(s-1)$ comparison operations. However, all points in S have the same value in the ND-cache. The ND-cache guarantees that the results of all $s(s-1)$ comparison operations are non-dominated.

Table 1 History of the skyline

Time	Skyline
T_0	$\{e_0, e_1\}$
T_1	$\{e_0, e_1, e_2, e_3\}$
T_2	$\{e_5\}$
T_3	$\{e_0, e_1, e_6\}$

Table 2 Contents of ND-cache at time T_3

Index	0	1	2	3	4	5	6
Value	T_3	T_3	T_1	T_1	null	T_2	T_3

The timestamp at which entry e_x last belonged to the skyline before T_3 is stored as a value at index x

Table 3 Complexity of the `dominates()` function

Complexity	Time	Space
No cache	$O(d)$	None
Cache all	$O(1)$ when hit	$O(N^2)$
ND-cache	$O(1)$ when hit	$O(N)$

The ND-cache stores information about former dominance relation calculations in a one-dimensional array instead of a set. This feature realizes the following advantages: (1) the later skyline is preferentially stored; (2) the old skyline, which has dropped into a deeper position, is maintained and not overwritten; and (3) the ND-cache is easily implemented.

The dominance relation includes both dominated and non-dominated relations. The BJR-tree stores the dominated relations, and the ND-cache stores the non-dominated relations between two skyline points. The BJR-tree and the ND-cache complement each other. However, the ND-cache is applicable not only to BJR-trees but also to other continuous skyline computation algorithms.

5 Complexity analysis

We first describe the spatial complexity of our BJR-tree. Each vertex in the BJR-tree contains the information of its parent and a variable-length list of its children. The spatial cost of the fixed information is $O(1)$ per vertex; thus, the total cost is $O(n)$, where n is the number of points that are active at the same time. Therefore, the spatial complexity of the BJR-tree is $O(n)$.

We now compute the complexities of the existing algorithms and our algorithm on a 2-dimensional dataset. The time complexity is the number of comparison operations of the dominance relation (which constitute the most expensive part of the algorithm). The average number of skyline points in a set of n independent d-dimensional vectors $A(n, d)$ is given by [11]

$$\frac{1}{(d-1)!}\left(\sum_{k=1}^{n}\frac{1}{k}\right)^{d-1} \leq A(n,d) \leq \left(\sum_{k=1}^{n}\frac{1}{k}\right)^{d-1} \tag{3}$$

When $d = 2$, we have $A(n, 2) = H(n)$, where $H(n)$ denotes the n-th harmonic number. In the left panel of Fig. 8, the number of children of the root is the same as the number of

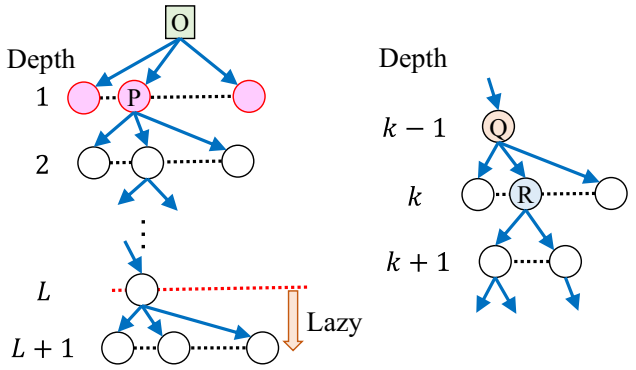

Fig. 8 Examples of BJR-trees built by the lazy algorithm at depths below L (left). When a vertex R is ejected, its children are moved to below vertex Q (right)

skyline points $H(n)$. Because the numbers of children in the BJR-tree are balanced, the average number of descendants of the vertices at depth 1 (e.g., vertex P), denoted by $D_1(n)$, is $\frac{n}{H(n)}$. Therefore, the number of children of a depth-1 vertex, $C_1(n)$, is $A(D_1, 2) = H\left(\frac{n}{H(n)}\right)$. These quantities meet the following criteria.

$$D_0(n) = n, \quad C_0(n) = H(n) \tag{4}$$

$$C_i = H(D_i), \quad D_k \prod_{i=0}^{k-1} C_i = n \tag{5}$$

We first describe the complexity of a single injection, which causes a recursive traversal of the BJR-tree. The dominance relations between the children of the root and the injected vertex are calculated at depth k with complexity $O(C_{k-1}(n))$. Therefore, the complexity of a single injection is

$$\sum_{i=0}^{d} C_i = C_0(n) + C_1(n) + \cdots + C_d(n) \tag{6}$$

$$= O(\log n) + O(\log n - \log\log n) + \cdots \tag{7}$$

$$= O(\log n) \tag{8}$$

We next show the time complexity of a single ejection. When a vertex R at depth k is ejected (Fig. 8, right), all of R's children (numbering $C_k(n)$) are injected into R's parent Q. If the depth of R equals or exceeds L, no dominance relation is calculated by the lazy evaluation; otherwise, the complexity of each injection is $O(\log D_{k-1})$. The time complexity of a single ejection without lazy evaluation is

$$C_k \cdot O(\log D_{k-1}) = O(H(D_k) \log D_{k-1}) \tag{9}$$

$$= O(\log D_k \log D_{k-1}) \tag{10}$$

$$= O\left(\log^2 n\right) \tag{11}$$

In the BBS method [42], the average complexity of both injection and ejection is $O(\log^2 n)$. In LookOut [40], the average complexities of injection and ejection are $O(\log n)$ and $O(\log^2 n)$, respectively, if the ejection of the LookOut consists only of the computation of dominance relations between the vertices. Actually, when traversing an R*-tree or quadtree, LookOut additionally needs the comparison operations between the ejected point and the maximum corner of the regions expanded in a heap. The worst-case cost of these operations is $O(n)$. Therefore, the BJR-tree incurs lower injection and ejection costs than BBS and LookOut, and more rapidly ejects the vertices deeper than L. Asymptotically, the complexity of the BJR-tree is the same as or better than those of previous methods.

In the worst-case injection scenario, when all active points belong to the skyline, the complexity of single injection into a BJR-tree is $O(n)$. In the worst-case ejection scenario, when the ejected point is the only skyline point and all others belong to the new skyline, the complexity is $O(n^2)$. The worst-case complexities of BJR-tree match that of LookOut.

6 Experimental evaluation

We now compare our proposed algorithm with two existing algorithms using two types of datasets: randomly generated synthetic datasets and datasets of real-world cell measurements. Datasets for continuous skyline computation contain both spatial and temporal information, i.e., the vector values of the points and their activation/deactivation times, respectively.

6.1 Synthetic datasets

We generated three types of random synthetic dataset, characterizing the spatial information by the correlation strengths of the spatial distributions. In a **correlated dataset (CORR)**, any point A that dominates another point B in one dimension tends to dominate B in other dimensions. All pairs of criteria for these points are strongly positively correlated. In an **independent dataset (INDE)**, each multi-dimensional vector is randomly determined and there are no correlations among dimensions. In an **anti-correlated dataset (ANTI)**, any point A that dominates another point B in one dimension tends to be dominated by B in the next dimension. In this dataset, the $2k+1$-th and $2(k+1)$-th axes are strongly negatively correlated. CORR datasets are easily handled because of their low skyline ratio, whereas ANTI datasets are well known to be difficult to handle [48]. The spatial features of static skyline computations are categorized in [9]. In the synthetic datasets of this study, the activation time $act(\boldsymbol{v})$ and lifetime $(deact(\boldsymbol{v}) - act(\boldsymbol{v}))$ of each point $\boldsymbol{v}$ were randomly selected.

The synthetic datasets were generated from simple parameters. To provide a pathological benchmark, we created

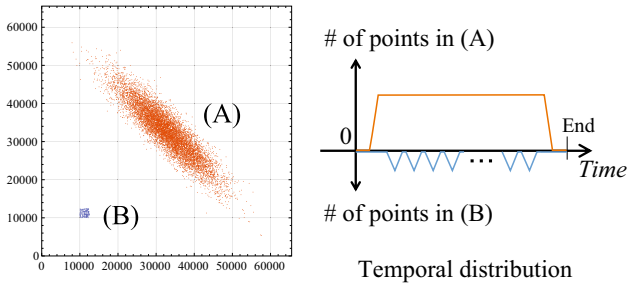

Fig. 9 Model of CONC. Set (A), with a strongly anti-correlated spatial distribution, is activated at the start and deactivated at the end. The points in set (B), which dominate most of the points in (A), are repeatedly activated and deactivated. At each activation (deactivation), some of the points in (A) are purged from (readmitted to) the skyline. These operations require heavy computational effort

Table 4 Synthetic dataset parameters

Cardinality	10k, 20k, 40k, . . ., 320k
Dimensionality	2, 3, 4, . . ., 25
Spatial distribution	Anti-correlated ($r = -0.9$), Independent ($r = -0.0$), Correlated ($r = +0.9$), Concealed
Temporal distribution	Independent, Concealed
Spatial value	[0, 65535]
Temporal value	[0, 49999]
Lifetime	[1000, 1999]

additional synthetic datasets with a drastically changing skyline. This dataset, called the concealed-skyline dataset (CONC), is shown in Fig. 9. Previous skyline points repeatedly leave and then return to the skyline, forcing the same dominance relations to be repeatedly recalculated. This dataset consists of a set of anti-correlated points (A) and multiple sets of independent points near the origin (B). When only (A) is activated, the skyline is a subset of (A), but when a set in (B) is activated, most of the skyline points in (A) are dominated. Then, when the set in (B) is deactivated, the points that were previously part of the skyline return to it. Such a dataset tends to require longer processing time than other datasets consisting of similar numbers of points.

The generated datasets contained up to 320,000 points in spaces of dimensions 2–25, and their parameters are shown in Table 4. For all criteria, smaller values are better, i.e., points with smaller values dominate those with larger ones.

6.2 Real cell measurement datasets

We then extracted features from cell images taken under a fluorescence microscope and created real-world datasets by using the feature vectors. The latencies of the proposed and existing methods were compared on these real-world datasets. Three types of real-world datasets are described in detail below.

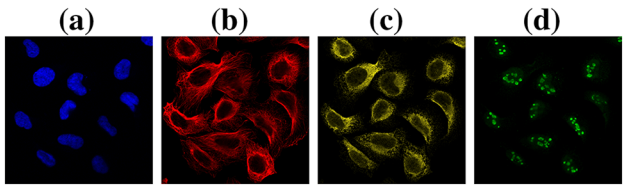

Fig. 10 Human protein cell images from the CYTO 2017 Image Analysis Challenge. Each image is captured in four channels: **a** DAPI for the nucleus, **b** antibody-based staining of microtubules, **c** endoplasmic reticulum, and **d** protein of interest

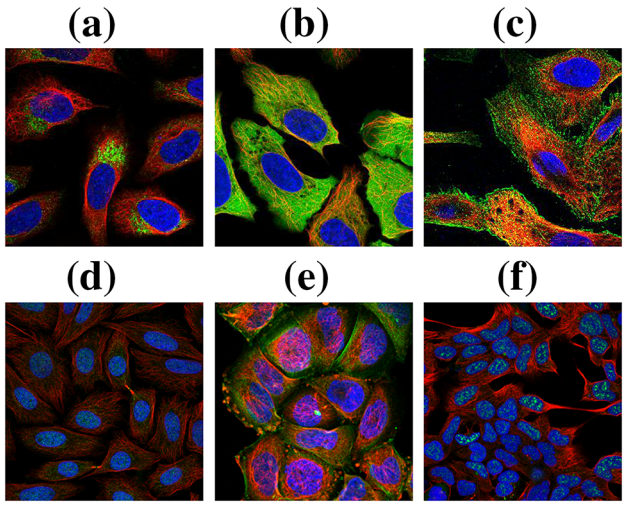

Fig. 11 Examples of the tagged cell image: **a** "Mitochondria," **b** "Cytosol," **c** "Plasma membrane," **d** "Cytokinetic bridge," **e** "Focal adhesion sites," and **f** "Nuclear speckles." In **a**–**c**, the labels indicate the protein location; in **d**–**f**, they indicate rare events in the cells

6.2.1 Human protein cell images

High-resolution images were taken from the CYTO 2017 Image Analysis Challenge [17]. These images, stored in the Human Protein Atlas database, were taken by a Leica SP5 confocal fluorescence microscope. They show immunostained human proteins in four fluorescence channels (see Fig. 10): (a) DAPI for the nucleus (blue), (b) antibody-based staining of microtubules (red), (c) endoplasmic reticulum (yellow), and (d) protein of interest (green). Each image was tagged with 19 types of labels, 13 indicating the protein location and six indicating rare events in the cells. For example, images (a), (b), and (c) in Fig. 11 were tagged with the protein-location labels "Mitochondria," "Cytosol," and "Plasma membrane," respectively. Images (d), (e), and (f) were tagged with the rare-event labels "Cytokinetic bridge," "Focal adhesion sites," and "Nuclear speckles," respectively.

The cell positions in the images are synchronized among the channels.

We first identified the nucleus region in the DAPI nuclear staining images. Second, we identified the entire cell region using the antibody-based microtubule staining images. Third, we identified the cytoplasm region by subtracting the nucleus region from the whole-cell region. In this way, we found 19,495 cells in 2,538 microscopic images. For each cell, we calculated a 1,014-dimensional feature vector including 48 area-shape features, 60 intensity features, 18 location features, and 212 texture features. We then constructed pipelines and analyzed these cells using the Cell Profiler [12]. We selected seven labels, namely "Euglena," "Mitochondria," "Nucleoli," "Cytosol," "Nucleus," "Aggresome," "Cytokinetic bridge," and "Focal adhesion sites," and individually assigned them to seven datasets, each consisting of 19,495 three-dimensional entries with two-, three-, or four-dimensional vectors. The condition positive rate in each dataset ranged from approximately 4.9 to 54.1%. All skyline entries in each dataset were tagged with each label (true condition). Thus, the skyline computation can identify the required entries in these biologically important datasets, confirming the suitability of these datasets for the performance evaluation. To create datasets for continuous skyline computation, we allocated uniformly distributed lifetimes to the entries. The maximum time step was set to 10,000.

6.2.2 Euglena cell images

We also used low-resolution cell images taken under a fluorescence microscope in flow cytometry (our Serendipiter [22]). The cells were *Euglena* cells, which are expected to be highly effective for biofuel production. The cells were captured in one transmitted light and two fluorescence channels (Fig. 12a). The images show lipid and chlorophyll staining of cells. The cell positions in the images are synchronized among the channels, and there is one *Euglena* cell per image. Each cell was tagged with one of two labels depending on the cell's environment: nitrogen deficient ("N-def") or nitrogen sufficient ("N-suf"). High-efficiency biofuel production will require the extraction and cultivation of *Euglena* cells that are lipid-rich in nitrogen deficient environments.

The goals of Serendipiter are simultaneous high image quality and throughput of cell flow. As the Serendipiter microscopes are still being developed, the images are noisy with low resolution and difficult to analyze in Cell Profiler. Instead, we analyzed the images ourselves. First, we identified the *Euglena* cell region and constructed mask images from the transmitted light images (Fig. 12b). In this way, we found 1072 *Euglena* cells in 1072 microscopic images. Second, we calculated a 47-dimensional feature vector for each cell, including 23 area-shape features from the mask images and 24 intensity features from the three-channel

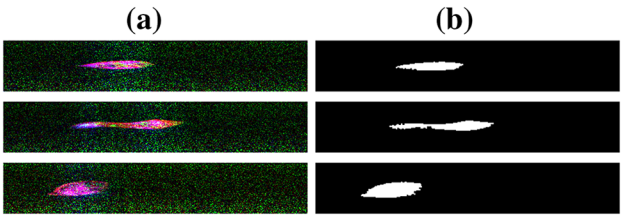

Fig. 12 **a** Images of *Euglena* cells under a fluorescence microscope in flow cytometry (Serendipiter). Each image is captured in three channels: transmitted light, lipid fluorescence, and chlorophyll fluorescence. **b** Mask images of the *Euglena* cell generated from the transmitted light channel

images. We then selected the features "IntensityMinimum' and "IntensityMean" in the transmitted light channel and "IntensityStandardDeviation" in the chlorophyll channel. Finally, we obtained the EUGLENA dataset consisting of 1072 three-dimensional entries. Approximately 7.4% of the entries in EUGLENA, and all of the skyline entries in EUGLENA, were tagged with "N-def." That is, the skyline computation extracted the rare "N-def" entries with 100% precision. To create datasets for continuous skyline computation, we allocated uniformly distributed lifetimes to the entries.

6.2.3 Blood cell images

We also generated datasets of blood cell measurements. The blood cell images contained aggregated platelets, single platelets, and white blood cells. In total, 4992 cell images were captured by optofluidic time-stretch quantitative phase microscopy, and the cell features were extracted from the images by Cell Profiler [12]. Cell Profiler is an open-source software that analyzes cell images for biologists. The cell information was provided by the authors of [26]. Figure 13 shows the heat map of the pairwise correlation matrix of 41 features (14 area-shape features, 13 intensity features, and 14 texture features) extracted from the blood cells. Ten of the area-shape features were strongly correlated with the "Area" feature (enclosed in the upper-left black box in Fig. 13). The intensity and texture features were also strongly correlated. As poorly correlated features in datasets are preferred for experiments, we eliminated the strongly correlated features and selected four weakly correlated features: the "Area" feature, one of the three area-shape features ("Compactness," "Eccentricity," or "Orientation"), one of the 13 intensity features, and one of the 14 texture features. Thereby, we created 546 types of datasets, each containing 4992 four-dimensional vectors. Figure 14 shows the cumulative distribution function of the maximum skyline ratio in these datasets. The skyline ratio was below 7% in all datasets and below 1% in 20% of the datasets. Therefore, these datasets were pragmatic.

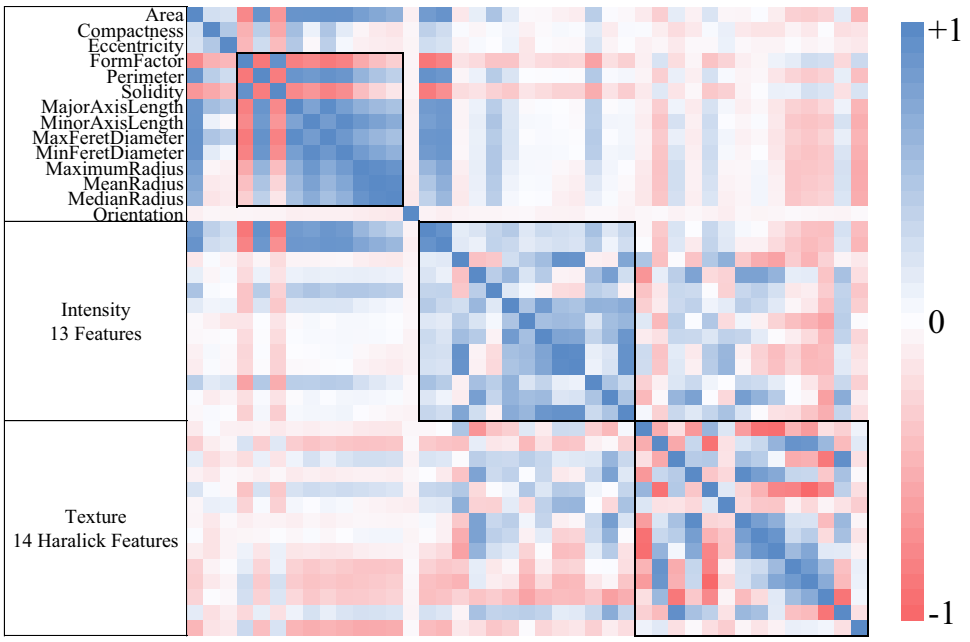

Fig. 13 Heat map displaying the correlations between 41 features obtained by Cell Profiler (blue and red indicate positive and negative correlations, respectively). The black boxes enclose the strongly correlated feature groups

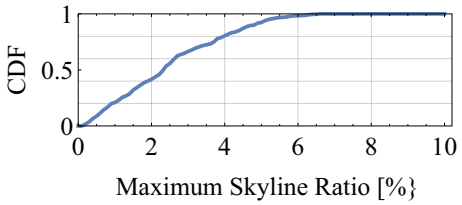

Fig. 14 Cumulative distribution function of the maximum skyline ratio in 546 real datasets. Each dataset contains 4992 four-dimensional vectors

Table 5 Specifications of our platform

Processor	Intel Core i7 6700K @ 4.0 GHz
Motherboard	ASUS Z170M-PLUS
Main memory	DDR4-2133 (8GB × 4)
Operating System	CentOS Linux 7.1.1503 x86_64
C/C++ compiler	GNU GCC 4.8.3

6.3 Experimental setup

We implemented the BJR-tree on an Intel x86_64 architecture in a single Intel-based computer. Although the processor was multi-cored, we executed and benchmarked all implementations in a single thread. The specifications are detailed in Table 5.

Our algorithm was competed against three existing algorithms. **Continuous BNL (cBNL)** and **Continuous BBS (cBBS)** extend the static skyline computation algorithms BNL [9] and BBS [42], respectively, to continuous skyline computation. **LookOut** is the state-of-the-art algorithm for continuous skyline computation [40]. All algorithms were implemented in C or C++. In all implementations, we employed the same interface functions to read the input datasets and output the skylines, and the same comparison functions for the dominance relations. The cBBS and LookOut algorithms were implemented on quadtrees, which accommodate a maximum of q points in their leaf regions.

6.4 Tree analysis

In Algorithm 3, we included a depth parameter L that controls the timing of the lazy evaluation and balances the tree. The relationship between execution time and L for different datasets is shown in Fig. 15. The cases of $L \to \inf$ and $L = 1$ are referred to as the *no-lazy* and *full-lazy* policies, respectively. The best L depended on the dataset and equaled 1 in the two-dimensional datasets. L exerted little influence on the execution time in the other datasets. Therefore, we adopted the full-lazy policy.

We now show the features of the constructed quadtrees and BJR-trees. Figure 16 shows the relationship between the cardinality and heights of the quadtrees and BJR-trees in independent datasets with two and eight dimensions. The trees were of four types; a BJR-tree with a full-lazy policy (blue), a BJR-tree with a no-lazy policy (yellow), a quadtree with $q = 1$ (green), and a quadtree with $q = 40$ (red). The large-q quadtree was shallower than the small-q quadtree, but the local skylines needed to be computed in each leaf region. Meanwhile, the height of the BJR-tree was lowered by the lazy evaluation. As shown in Fig. 16 (right), in the eight-dimensional dataset, the heights of all

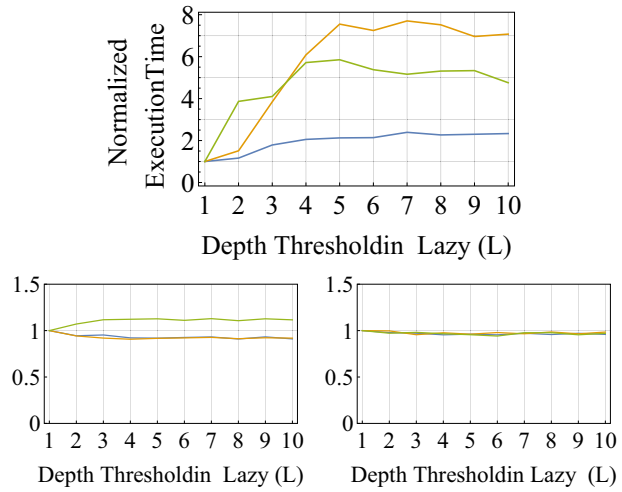

Fig. 15 Execution time (normalized such that $y = 1$ when $L = 1$) versus depth threshold L in the BJR-tree algorithm for different datasets with 320k cardinality: ANTI (blue), INDE (orange), CORR (green), dim=2 (top), dim = 5 (bottom-left), and dim = 8 (bottom-right)

four trees were very similar. The number of spatial partitions in the spatial-indexing tree exponentially increased with increasing number of dimensions. Establishing a dominated relation between two independent points is more difficult in a high-dimensional space than in a low-dimensional space, which explains the lack of height difference in the eight-dimensional dataset. Therefore, the tree structure largely depends on the number of dimensions. It is valuable for low-dimensional datasets to lower the height of the trees.

Panel (a) of Fig. 17 shows the skyline potential of the two-dimensional vectors, and panel (b) displays the depths of the vertices corresponding to each vector in the BJR-tree expressing the same two-dimensional vectors. In each distribution, the BJR-tree expressed the skyline potential of each vector as a vertex depth in the tree. Moreover, the skyline potential and vertex depth were highly consistent near the origin (in the high skyline potential area). The BJR-tree stores the skyline potentials as the vertex positions.

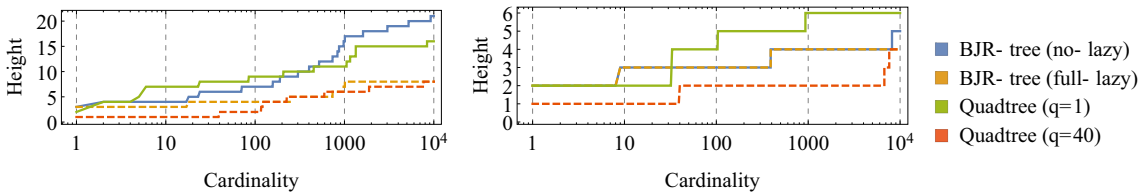

Fig. 16 Heights of the BJR-trees and quadtrees constructed from up to 10,000 points with two (left) and eight (right) dimensions. The green quadtree has a single point per leaf, and the red quadtree accommodates up to 40 points per leaf

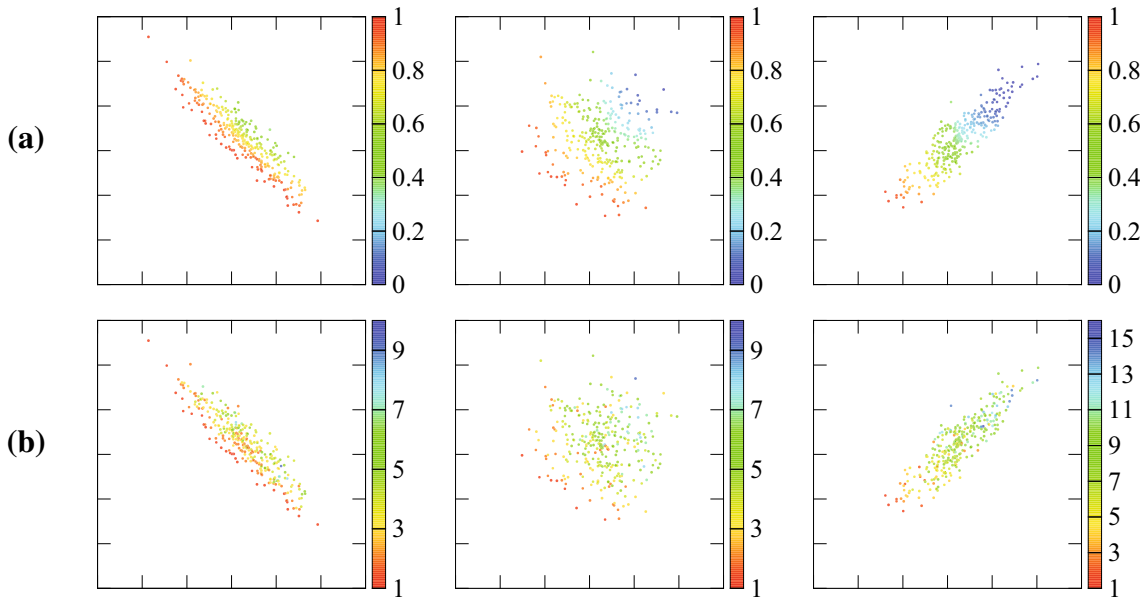

Fig. 17 **a** Skyline potentials of the points and **b** BJR-tree depth positions of the corresponding vertices (left: ANTI, center: INDE, and right: CORR)

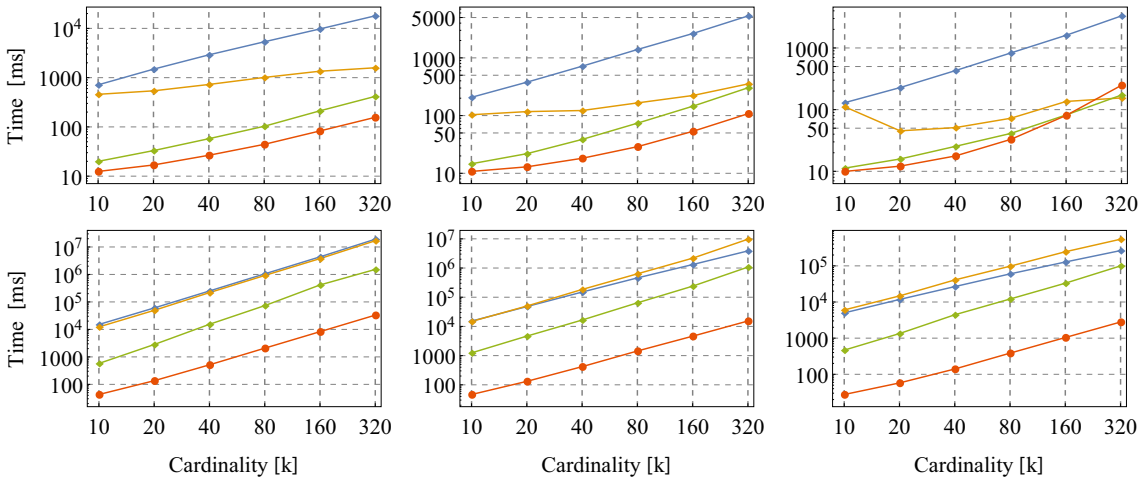

Fig. 18 Execution times versus dataset cardinality (left: ANTI, center: INDE, and right: CORR/upper: 2D and lower: 8D) in different algorithms: cBNL (blue), cBBS (yellow), LookOut (green), and BJR-tree (red). All vertical axes denote execution time in milliseconds

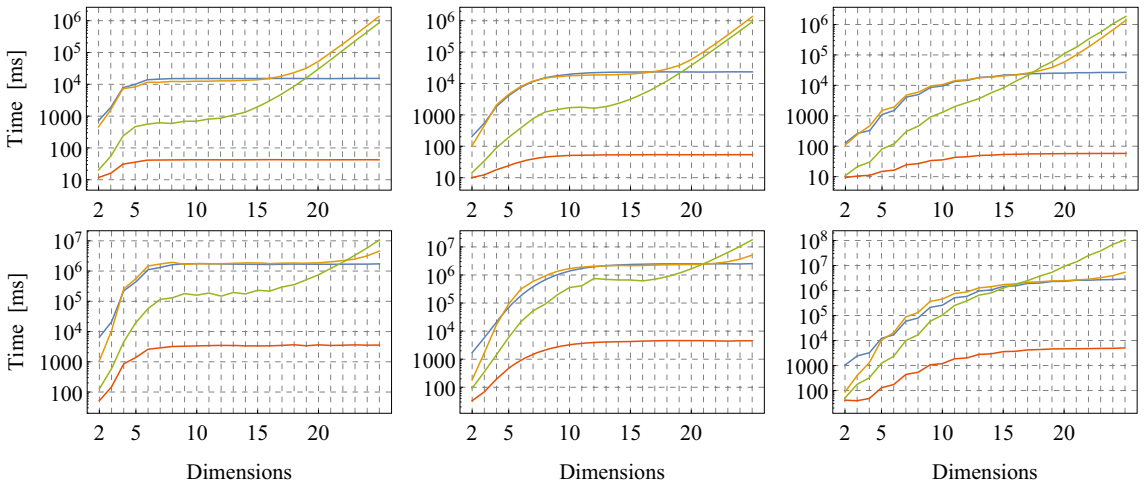

Fig. 19 Execution time versus number of dimensions (left: ANTI, center: INDE, and right: CORR/upper: 10k and lower: 100k) in different algorithms: cBNL (blue), cBBS (yellow), LookOut (green), and BJR-tree (red). All vertical axes denote execution time in milliseconds

6.5 Results for synthetic datasets

We evaluated the execution times of our proposed algorithm and other existing continuous skyline computation algorithms on the synthetic datasets, excluding the times consumed by initialization, memory loading of the points from the input file, and storing the skylines from the array to an output file.

6.5.1 Cardinality

The execution times are plotted as functions of cardinality in Fig. 18. In this comparison, the dimensions are two or eight. On almost all datasets, our BJR-tree outperformed the existing algorithms; the only exception was the CORR dataset with 320,000 two-dimensional points, on which cBBS and Lookout performed 1.6 times faster than BJR-tree. On the INDE dataset with 320,000 eight-dimensional points, BJR-tree executed more than 70 times faster than LookOut.

6.5.2 Dimensionality

Figure 19 shows the relationships between the number of dimensions and the execution time at cardinalities of 10,000 and 100,000. The execution time only slightly varied with number of dimensions because cBNL does not use a tree structure and BJR-tree is dimensionally independent. The execution time of BJR-tree mostly depended on the skyline ratio (see Fig. 2). At lower dimensions, the execution times of cBBS and LookOut also depended mainly on the skyline ratio, but a dimensional effect emerged at up to 14 dimensions. On the CORR dataset with 25-dimensional

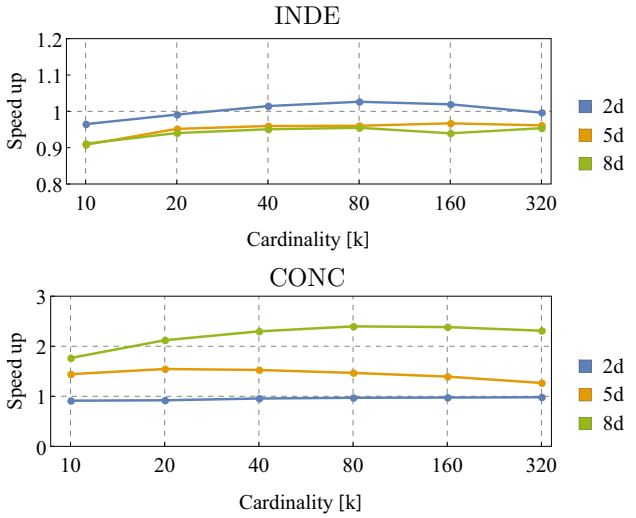

Fig. 20 Speed-up ratio of BJR-tree with an ND-cache

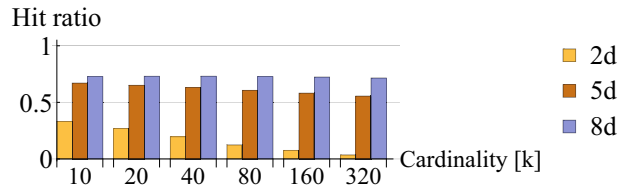

Fig. 21 Hit ratios of the ND-cache in CONC

points, the BJR-tree outperformed LookOut by a factor of approximately 570. For example, the skyline ratio of the eight-dimensional CORR was approximately 0.3. On the INDE dataset with 100,000 eight-dimensional points, BJR-tree was approximately 43 times faster than LookOut.

6.5.3 Concealed-skyline dataset

Processing CONC datasets is time-intensive because the ejection operations incur a heavy computational load. Our ND-cache resolves this problem. Figure 20 compares the execution times of BJR-trees with and without an ND-cache. The x-axis indicates the number of anti-correlated points [set (A) in Fig. 9]. The anti-correlated points were dominated by 200 temporal clusters [set (B) in Fig. 9], each with five points. The ND-cache reduced the CONC execution time without significantly increasing the execution time of the INDE datasets. In the CORR dataset with eight-dimensional points, the BJR-tree executed approximately 2.3 times faster with the ND-cache than without the cache. The ND-cache achieved a hit ratio of 50–70% in five- and eight-dimensional CONC datasets (see Fig. 21).

6.6 Results for real-world cell measurement datasets

Finally, we competed our proposed algorithm against three existing algorithms for continuous skyline computation on real-world datasets. Figure 22 shows the speed-up ratios calculated from the execution times of the existing method implementations (cBNL, cBBS, and LookOut), and our BJR-tree implementation for eight datasets. On average, the BJR-tree reduced the execution time by 97.7% from that of cBNL, 94.4% from that of cBBS, and 63.9% from that of LookOut. On the Euglena dataset and Mitochondria datasets, BJR-tree executed 2.1 and 9.7 times faster than LookOut, respectively.

Figure 23 shows the execution times of the algorithms on real-world datasets of blood cells. Plotted are the cumulative distribution functions of 546 blood cell datasets. In real-world datasets, the algorithm performance depends on the feature selection. The average execution times of BJR-tree and LookOut were 14.2 and 33.8 μs, respectively. In almost all datasets, the execution time of BJR-tree was below 20 ms. In contrast, LookOut required over 100 ms in 17 datasets, and cBNL and cBBS required over 1 s in almost all datasets. Therefore, in both synthesized and real-world datasets, the execution time of BJR-tree surpasses the execution times of existing algorithms.

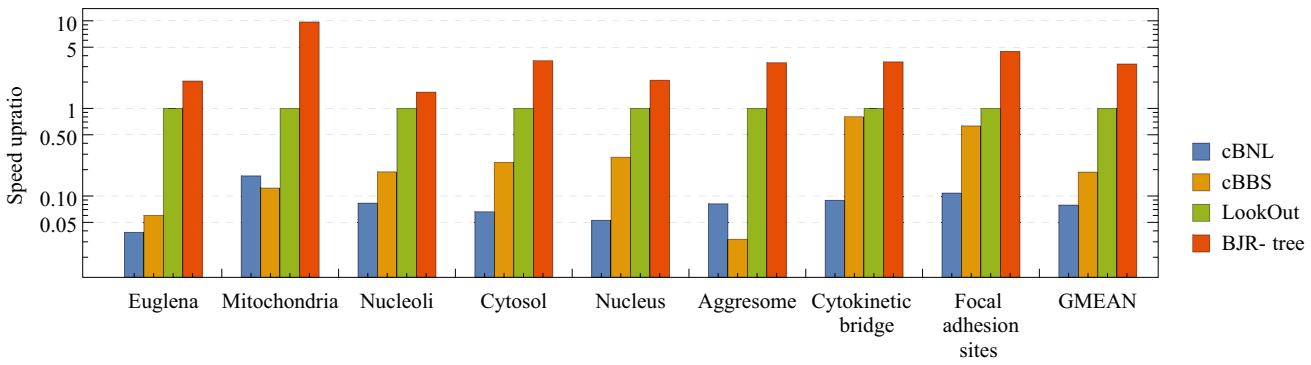

Fig. 22 Speed-up ratios in the existing methods (cBNL, cBBS, LookOut), and our BJR-tree implementation for eight datasets. In each dataset, the speed-up ratio (y-axis) of the four methods is normalized such that the LookOut value is 1

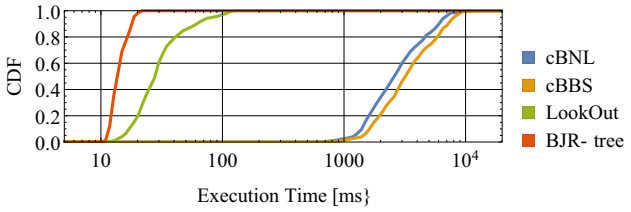

Fig. 23 Cumulative distribution function of the execution times in real-world blood cell datasets

7 Discussion

In this section, we discuss performances of skyline operation and other entry extraction methods on real-world applications. We also describe another application of the proposed BJR-tree.

Gating is a simple method for extracting the target cell entries in cell information analysis. This statistical method is widely used in cell analysis. After selecting two or more cell measurement metrics, the measurement data are projected to a space with two or more dimensions. The minimum and maximum thresholds in each dimension are then set, and the subspace enclosed by the thresholds is obtained. By filtering the entries outside the enclosed subspace, the purity of the measurement data is improved. However, gating is problematic for two reasons. First, the metrics must be appropriately selected to isolate sets of cells. Unless we know the metrics that can divide cell clusters, gating is inapplicable to cell identification. Another problem is the low precision of gating. Gating extracts the large-scale populations in low-dimensional spaces. The aim of the gating method is to improve the purity of a relatively large-scale population. Therefore, the gating method is not suitable to improve the precision of extracting extremely rare cells. For example, in the datasets of the CYTO 2017 Cell Image Analysis Challenge, the proteins in cells tagged with "Cytosol" and "Endoplasmic reticulum" occupy intracellular regions outside the cell nucleus. Therefore, in the cell images, the channel intensity of the protein is high in the cytosol and endoplasmic reticulum, respectively. Meanwhile, the proteins in cells tagged with "Nuclear membrane" reside at the boundary between the cell nucleus and the surrounding region. In these cell images, the channel intensity of the protein is high at the outer perimeter of the cell nucleus. Many tagged cells can be extracted by combining metrics that appropriately describe the cell characteristics. However, this rough approach is unsuitable for extracting cells with rare labels. Thus, gating is an ineffective method for extracting rare cells in the Serendipiter.

Clustering methods, including gating, are employed in the extraction of large-scale populations. In the data analysis domains, researchers have applied outlier detection methods to rare-entry extraction. Outlier detection is also termed anomaly detection. In some applications of Serendipiter, obtaining large amounts of measured samples tagged with a rare label and creating a model from the extracted data are difficult tasks. To alleviate these difficulties, we often employ unsupervised machine learning in the Serendipiter. Typical unsupervised outlier detection algorithms are based on the local outlier factor (LOF) [10]. In addition, a method based on the one-class support vector machine (OC-SVM) [46] is also used. In high-dimensional spaces, the entries are very sparsely distributed owing to the curse of dimensionality, so the outliers are difficult to detect. Consequently, the LOF-based and other distance-based outlier detection methods are unsuitable for high-precision rare-entry extraction in high-dimensional space. This difficulty in high-dimensional space has been overcome by the angle-based outlier detection (ABOD) method [33]. ABOD and other angle-based methods assume that the variance of angles among vectors of an outlier entry to other entries is low even in high-dimensional space. However, unsupervised learning methods, even when robust in high-dimensional datasets, cannot easily extract entries with a rare label.

Outlier detection methods are effective because they require no preselection of the dimension. However, in real-time operations, we have to reduce the number of dimensions and implement a high-speed identification algorithm. The cells to be discovered are neither outliers nor anomaly cells, but are regular cells with extreme characteristics or rare labels. Therefore, outlier detection methods are inappropriate for our purpose. Instead, our problem can potentially be solved by skyline computation, which was originally proposed for extracting interesting entries. Rare-labeled cells on the Serendipiter are expected to be poorly identified by unsupervised machine learning with no dimensional reduction. Furthermore, the appropriateness of an identification algorithm depends on the cell type. In this study, we discussed the effectiveness of the combination of skyline computation and appropriate dimension reduction. We also showed the robustness of our algorithm to the low-latency real-time query processing of continuous skyline computation.

The BJR-tree is robust in high-dimensional spaces. As the number of dimensions increases, the skyline ratio approaches 1, and we cannot obtain an effective subset of entries by using the skyline computation. Even in datasets with same cardinality, the skyline ratio varies greatly because of correlation coefficient of the entry distribution. The skyline ratio of a dataset with positive or negative correlations is higher and lower, respectively, than that of an uncorrelated dataset. Nonetheless, as shown in Fig. 2, in synthetic datasets with 25 or more dimensions and practical correlation values, almost all entries join the skyline. In this study, we showed that BJR-tree operates faster than the existing algorithms in datasets with up to 25 dimensions. In both the experimental

results and theoretical analysis of computational complexity, it is showed that the execution time of BJR-tree in high-dimensional datasets strongly depended on the number of skyline entries. Thus, even in real-world datasets, the number of skyline entries and the execution time of BJR-tree can be controlled by adjusting the window size of the computational target entries.

The BJR-tree operates efficiently in continuous skyline computation. We now present another application other than continuous skyline computation of the BJR-tree. Because the skyline ratio is uniquely determined relative to a dataset. Therefore, skyline computation cannot be used for extracting an arbitrarily sized subset of entries. In addition, the skyline ratio largely depends on the distribution of entries in a multi-dimensional space. Thus, we cannot stably extract an entry subset of the expected size. Previously, entries surrounding the skyline have been extracted by methods such as skyband [43] and a distance-based method [49], which use geometric information (e.g., the norm). As mentioned above, when the number of dimensions is high, the entries are sparsely distributed and cannot be successfully extracted. The concept of k-dominant skyline [14] also provides another extraction method with dimension selection. However, it is necessary to compute a skyline in each combination of dimensions. It appreciably increases the computational time. To address this problem, we can employ the skyline potential concept. All entries in a dataset can be ranked based on a value of their skyline potential. Because the time complexity of obtaining the exact values of skyline potential of all entries is $O(N^2)$, we can instead construct a BJR-tree and roughly rank the entries based on the depths of their corresponding nodes. Thus, only by a single construction of the BJR-tree, we can obtain entry subsets of various sizes. This application exemplifies how entry extraction methods based on the BJR-tree can compensate the weak points of skyline operation.

8 Conclusion

The skyline computation algorithm is used for extracting the interesting entries from a database of multi-attribute entries. The task of computing a *dynamic* set of points is known as the continuous skyline computation. The continuous skyline computation is useful for removing the non-skyline points in preprocessing for screening a large amount of data in real time. We proposed the BJR-tree structure for continuous skyline computation. The fast speed of the BJR-tree is conferred by an appropriate hierarchical expression and dimensionality independence. To handle artificial datasets with temporal features requiring many computations, we proposed an ND-cache mechanism. The BJR-tree and ND-cache store the dominated relations and the important non-dominated relations, respectively.

We competed our proposed algorithm against the extended BNL and BBS (for continuous skyline computation), and LookOut. We used datasets with randomly generated vectors and real datasets of blood cell measurements. On the synthetic datasets, the BJR-tree computed the continuous skylines approximately 3–70 times faster than LookOut. On real-world datasets, the BJR-tree is approximately 2.4–3.2 times faster than LookOut.

Our main contributions are as follows: (a) We proposed a new tree structure and a new cache mechanism for continuous skyline computation. Our approach reduces the number of avoidable comparisons and stores the previously calculated results. (b) Combined with the ND-cache, our proposed algorithm speeds up the continuous skyline computation, as confirmed in comparisons with existing algorithms. (c) In terms of execution time, BJR-tree outperforms LookOut on real-world medium-dimensional datasets extracted from cells.

To determine the usefulness of the skyline cells extracted by BJR-tree, we must biochemically analyze the cells after repetitive cultivation and extraction. This paper confirmed that by appropriately selecting the features, we can stably isolate rare skyline cells.

Acknowledgements This work was partially funded by ImPACT Program of Council for Science, Technology and Innovation (Cabinet Office, Government of Japan). We would like to acknowledge Dr. Lei, Dr. Ozeki, Dr. Sugimura, and Dr. Goda for providing measurement results of blood cells. We thank H. Tezuka for constructive comments.

References

1. Bartolini, I., Ciaccia, P., Patella, M.: Efficient sort-based skyline evaluation. ACM Trans. Database Syst. **33**(4), 31:1–31:49 (2008)
2. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. Acta Inf. **1**(3), 173–189 (1972)
3. BD Biosciences: Cell Sorters. http://www.bdbiosciences.com/us/instruments/research/cell-sorters/c/744762
4. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, SIGMOD '90, pp. 322–331. ACM, New York, NY, USA (1990)
5. Berchtold, S., Keim, D.A., Kriegel, H.P.: The X-tree: an index structure for high-dimensional data. In: Proceedings of the 22th International Conference on Very Large Data Bases, VLDB '96, pp. 28–39. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
6. Bøgh, K.S., Assent, I., Magnani, M.: Efficient GPU-based skyline computation. In: Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN '13, pp. 5:1–5:6. ACM, New York, NY, USA (2013)
7. Bøgh, K.S., Chester, S., Assent, I.: Work-efficient parallel skyline computation for the GPU. Proc. VLDB Endow. **8**(9), 962–973 (2015)
8. Böhm, C., Kriegel, H.P.: Determining the convex hull in large multidimensional databases. In: Data Warehousing and Knowledge Discovery, pp. 294–306. Springer, Berlin (2001)

9. Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: Proceedings 17th International Conference on Data Engineering, pp. 421–430 (2001)
10. Breunig, M.M., Kriegel, H.P., Ng, R.T., Sander, J.: LOF: identifying density-based local outliers. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, pp. 93–104. ACM, New York, NY, USA (2000)
11. Buchta, C.: On the average number of maxima in a set of vectors. Inf. Process. Lett. **33**(2), 63–65 (1989)
12. Carpenter, A.E., Jones, T.R., Lamprecht, M.R., Clarke, C., Kang, I.H., Friman, O., Guertin, D.A., Chang, J.H., Lindquist, R.A., Moffat, J., Golland, P., Sabatini, D.M.: Cell Profiler: image analysis software for identifying and quantifying cell phenotypes. Genome Biol. **7**(10), R100 (2006)
13. Chan, C.Y., Jagadish, H., Tan, K.L., Tung, A.K., Zhang, Z.: Finding k-dominant skylines in high dimensional space. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 503–514. ACM (2006)
14. Chan, C.Y., Jagadish, H.V., Tan, K.L., Tung, A.K.H., Zhang, Z.: Finding K-dominant skylines in high dimensional space. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06, pp. 503–514. ACM, New York, NY, USA (2006)
15. Choi, W., Liu, L., Yu, B.: Multi-criteria decision making with skyline computation. In: 2012 IEEE 13th International Conference on Information Reuse and Integration (IRI), pp. 316–323. IEEE (2012)
16. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: Proceedings of 19th International Conference on Data Engineering, pp. 717–719. IEEE (2003)
17. CYTO: CYTO2017 Image Analysis Challenge. http://cytoconference.org/2017/Home.aspx (2017)
18. Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta Inf. **4**(1), 1–9 (1974)
19. Fotiadou, K., Pitoura, E.: BITPEER: continuous subspace skyline computation with distributed bitmap indexes. In: Proceedings of the 2008 International Workshop on Data Management in Peer-to-Peer Systems, pp. 35–42. ACM (2008)
20. Godfrey, P., Shipley, R., Gryz, J.: Algorithms and analyses for maximal vector computation. VLDB J. Int. J. Very Large Data Bases **16**(1), 5–28 (2007)
21. Graham, R.L.: An efficient algorith for determining the convex hull of a finite planar set. Inf. Process. Lett. **1**(4), 132–133 (1972)
22. Guo, B., Lei, C., Kobayashi, H., Ito, T., Yalikun, Y., Jiang, Y., Tanaka, Y., Ozeki, Y., Goda, K.: High-throughput, label-free, single-cell, microalgal lipid screening by machine-learning-equipped optofluidic time-stretch quantitative phase microscopy. Cytom. A **91**(5), 494–502 (2017)
23. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD '84, pp. 47–57. ACM, New York, NY, USA (1984)
24. Hiraki, K., Inaba, M., Tezuka, H., Tomari, H., Koizumi, K., Kondo, S.: All-IP-ethernet architecture for real-time sensor-fusion processing. In: Proceedings of the SPIE, High-Speed Biomedical Imaging and Spectroscopy: Toward Big Data Instrumentation and Management, vol. 9720, p. 97200D (2016)
25. Huang, Z., Lu, H., Ooi, B.C., Tung, A.K.H.: Continuous skyline queries for moving objects. IEEE Trans. Knowl. Data Eng. **18**(12), 1645–1658 (2006)
26. Jiang, Y., Lei, C., Yasumoto, A., Kobayashi, H., Aisaka, Y., Ito, T., Guo, B., Nitta, N., Kutsuna, N., Ozeki, Y., et al.: Label-free detection of aggregated platelets in blood by machine-learning-aided optofluidic time-stretch microscopy. Lab Chip **17**(14), 2426–2434 (2017)
27. Katayama, N., Satoh, S.: The SR-tree: an index structure for high-dimensional nearest neighbor queries. ACM SIGMOD Rec. **26**(2), 369–380 (1997)
28. Kim, Y.J., Patel, J.M.: Rethinking choices for multi-dimensional point indexing: making the case for the often ignored quadtree. In: CIDR, pp. 281–291 (2007)
29. Koizumi, K., Eades, P., Hiraki, K., Inaba, M.: BJR-tree: fast skyline computation algorithm for serendipitous searching problems. In: 2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA) (2017)
30. Koizumi, K., Inaba, M., Hiraki, K.: Efficient implementation of continuous skyline computation on a multi-core processor. In: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 52–55 (2015)
31. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: an online algorithm for skyline queries. In: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 275–286. VLDB Endowment (2002)
32. Kothuri, R.K.V., Ravada, S., Abugov, D.: Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 546–557. ACM (2002)
33. Kriegel, H.P., S hubert, M., Zimek, A.: Angle-based Outlier Detection in High-dimensional Data. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '08, pp. 444–452. ACM, New York, NY, USA (2008)
34. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. JACM **22**(4), 469–476 (1975)
35. Lee, J., Hwang, S.W.: BSkyTree: scalable skyline computation using a balanced pivot selection. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 195–206. ACM (2010)
36. Lee, M.W., Hwang, S.w.: Continuous Skylining on Volatile Moving Data. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE '09, pp. 1568–1575. IEEE Computer Society, Washington, DC, USA (2009)
37. Liknes, S., Vlachou, A., Doulkeridis, C., Nørvåg, K.: APSkyline: improved skyline computation for multicore architectures. In: Database Systems for Advanced Applications, pp. 312–326. Springer (2014)
38. Lin, X., Yuan, Y., Wang, W., Lu, H.: Stabbing the sky: efficient skyline computation over sliding windows. In: Proceedings of the 21st International Conference on Data Engineering, ICDE '05, pp. 502–513. IEEE Computer Society, Washington, DC, USA (2005)
39. Milder, P.: MEMOCODE 2015 design contest: continuous skyline computation. In: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 48–51. IEEE (2015)
40. Morse, M., Patel, J.M., Grosky, W.I.: Efficient continuous skyline computation. In: 22nd International Conference on Data Engineering (ICDE'06), pp. 108–108 (2006)
41. Oikawa, M., Hiyama, D., Hirayama, R., Hasegawa, S., Endo, Y., Sugie, T., Tsumura, N., Kuroshima, M., Maki, M., Okada, G., Lei, C., Ozeki, Y., Goda, K., Shimobaba, T.: A computational approach to real-time image processing for serial time-encoded amplified microscopy. In: Proceedings of the SPIE, High-Speed Biomedical Imaging and Spectroscopy: Toward Big Data Instrumentation and Management, vol. 9720, p. 97200E (2016)
42. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 467–478. ACM (2003)
43. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. ACM Trans. Database Syst. **30**(1), 41–82 (2005)

44. Raj, P., Raman, A., Nagaraj, D., Duggirala, S.: High-Performance Big-Data Analytics: Computing Systems and Approaches, 1st edn. Springer, Berlin (2015)
45. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. ACM Sigmod Rec. **24**(2), 71–79 (1995)
46. Schölkopf, B., Platt, J.C., Shawe-Taylor, J., Smola, A.J., Williamson, R.C.: Estimating the support of a high-dimensional distribution. Neural Comput. **13**(7), 1443–1471 (2001)
47. Selke, J., Lofi, C., Balke, W.-T.: Highly scalable multiprocessing algorithms for preference-based database retrieval. In: Database Systems for Advanced Applications, pp. 246–260. Springer, Berlin (2010)
48. Shang, H., Kitsuregawa, M.: Skyline operator on anti-correlated distributions. Proc. VLDB Endow. **6**(9), 649–660 (2013)
49. Su, L., Zou, P., Jia, Y.: Adaptive Mining the Approximate Skyline Over Data Stream, pp. 742–745. Springer, Berlin (2007)
50. Tan, K.L., Eng, P.K., Ooi, B.C., et al.: Efficient progressive skyline computation. In: Proceedings of the 27th International Conference on Very Large Data Bases, vol. 1, pp. 301–310 (2001)
51. Tao, Y., Papadias, D.: Maintaining sliding window skylines on data streams. IEEE Trans. Knowl. Data Eng. **18**(3), 377–391 (2006)
52. Tian, L., Wang, L., Zou, P., Jia, Y., Li, A.: Continuous monitoring of skyline query over highly dynamic moving objects. In: Proceedings of the 6th ACM International Workshop on Data Engineering for Wireless and Mobile Access, pp. 59–66. ACM (2007)
53. White, D.A., Jain, R.: Similarity indexing with the SS-tree. In: Proceedings of the Twelfth International Conference on Data Engineering, ICDE '96, pp. 516–523. IEEE Computer Society, Washington, DC, USA (1996)
54. Woods, L., Alonso, G., Teubner, J.: Parallel computation of skyline queries. In: Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM '13, pp. 1–8. IEEE Computer Society, Washington, DC, USA (2013)
55. Woods, L., Alonso, G., Teubner, J.: Parallelizing data processing on FPGAs with shifter lists. TRETS **8**(2), 7:1–7:22 (2015)
56. Zhang, S., Mamoulis, N., Cheung, D.W.: Scalable skyline computation using object-based space partitioning. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 483–494. ACM (2009)