iTopN: Incremental Extraction of the N Most Visible Objects

Linas Bukauskas†, Leo Mark‡, Edward Omiecinski‡, Michael H. Böhlen†
linb@cs.auc.dk leomark@cc.gatech.edu edwardo@cc.gatech.edu boehlen@cs.auc.dk
†Department of Computer Science, Aalborg University, Denmark
‡College of Computing, Georgia Tech, Atlanta, Georgia, USA

ABSTRACT

The visual exploration of large databases calls for a tight coupling of database and visualization systems. Current visualization systems typically fetch all the data and organize it in a scene tree, which is then used to render the visible data. For immersive data explorations, where an observer navigates in a potentially huge data space and explores selected data regions this approach is inadequate. A scalable approach is to make the database system observer-aware and exchange the data that is visible and most relevant to the observer.

In this paper we present iTopN an incremental algorithm for extracting the most visible objects relative to the current position of the observer. We implement iTopN and compare it to an improved version of the R-tree that extends LRU with the caching of the top levels of the R-tree (LW-LRU). Our experiments show that iTopN is orders of magnitude faster than LW-LRU given the same amount of memory. Our experiments also show that for LW-LRU to perform as fast as iTopN it needs three times as much memory.

Categories and Subject Descriptors
H.2.8 [Database Management]: Database Applications—Data mining; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—Relevance feedback

General Terms
Algorithms, performance

Keywords
Indexing visibility ranges; Moving observer; Incremental observer relative data extraction; Top most visible objects

1. INTRODUCTION

For large-scale immersive data explorations it is essential to control the amount of data used in visualization scenes. Visualization systems currently use a static approach. At the beginning of a visual data exploration, the data is loaded into memory and a static scene tree is created. The visualization system uses this potentially huge scene-tree to support observer-controlled visual data explorations.

As an alternative, if a database system is used as a back-end for a visual data exploration system the extraction of the data that is relevant to an ongoing exploration could be performed incrementally. In order to turn a database system into an efficient back-end to a visualization system an intelligent data extraction algorithm is needed. When exploring data in a Panorama or Cave only a small portion of the data may be visible to an observer, and an even smaller portion may be highly visible to that observer. Therefore, we introduce an observer position relative data extraction function, which keeps track of the N most visible objects. When the observer moves through the data space it is important to provide a highly efficient, incremental algorithm for computing the N most visible objects.

In this paper we propose iTopN an incremental algorithm to extract the N most visible objects. We use the VR-tree, a refined version of the R-tree, to index the visibility ranges of objects. In order to further reduce and balance the I/O operations the VR-tree is associated with a novel volatile access structure (VAST). VAST is a main memory structure that caches traces of the VR-tree and reports appearance and disappearance of visible objects. A lightweight buffer supports the incremental management of the N most visible objects. We also describe our experimental evaluation of iTopN. The benefit of having VAST is that it reduces the number of I/O operations on the VR-tree. It allows fast pruning of invisible objects. Another benefit is that VAST grows and shrinks based on the amount of data, and does not stay full as in the case of traditional caching (i.e., LRU).

In summary, the main goals and contributions of this paper are:

- An efficient algorithm to store and incrementally maintain \( V^\text{TopN}_P \), the N most visible objects at observer position \( P \).
- An efficient way to compute appearing and disappearing objects, \( \Sigma^+_P, \Sigma^-_{P,P'} \), when the observer navigates in the data space.
- The empirical verification of the effectiveness and efficiency of the iTopN algorithm.

Many spatial database access structures are based on the R-tree [1, 4]. R-tree based structures use minimum bounding rectangles (MBRs) to hierarchically group objects and...
they support fast lookups of objects that overlap a query point/window. The R-tree has been extended in various directions to, e.g., index high-dimensional data, support similarity searches, or index moving objects [2, 11, 13, 14]. Another family of access structures is based on space partitioning as used in the kdB-tree [6, 8]. Common to all structures is a spatial grouping of objects. In our case the visible objects are not necessarily located in the same area. Each object has its own visibility range and therefore the visible objects may be located anywhere in the universe. In addition to the lack of a spatial grouping of visible objects the above mentioned access structures also do not support an incremental extraction of objects.

Related work has also been done in the area of k-nearest neighbor search for a moving query point [10]. The basic idea is to refine the nearest neighbor query results when the query point is moving. This is done by increasing the number of prefetched objects which permits restricted observer movement without re-querying database. The set of new nearest neighbors for the new query point is extracted using partially pre-fetched results that are based on the distance. The algorithm uses the pre-fetching to speed up the new search. In our setup we also consider the visibility range of an object. This is necessary because an object that is far from the observer might have a high visibility and, thus, be relevant. On the other hand a nearby object might not be relevant because of a low visibility.

The level of detail has been used extensively to optimize visualizations and visualization-based walk-through scenarios. Kofer et al. [7], Shou et al. [9] and Varadhan et al. [12] optimize massive visualization scenes by introducing the level of detail of extracted objects. Such a technique reduces the number of polygons used. Culling techniques are also used to optimize the search in R-tree like scene trees by exploiting view frustum and occlusions of objects. Our approach to extract objects based on their visibility measure is an orthogonal solution. Occlusion, visibility range, and view frustum can be combined to effectively reduce the number of rendered objects.

This paper is structured as follows. We start out with our application scenario in Section 2. Section 3 defines the problem. Section 4 describes the VR-tree and VAST, and discusses our implementation of the incremental extraction of all visible objects. In Section 4.4 we describe an extension of VAST that enables the incremental extraction of only the top N most visible objects. In Section 5 we present experimental results and compare the iTopN caching mechanism with an R-tree with level-wise caching. We finish the paper with conclusions and future work in Section 6.

2. APPLICATION SCENARIO

iToph has been developed in the context of the 3D Visual Data Mining (3VDM) System [3], which we use to visualize large data sets in immersive VR arenas (Panorama and Cave). The observer explores the data by navigating in the data space. At the technical level real time interaction becomes infeasible if more than 100’000 objects (displayed as tetrahedrons) have to be rendered. Clearly, this does not go well with the visualization of databases with several millions of observations. It is essential to extract only the visible objects. However, even the number of visible objects may be too large for state-of-the-art visualization devices to run efficiently. iToph has been developed to further reduce the number of visible objects to the N most visible objects from a given observer position.

Assume a visibility range that is proportional to the brightness of an object and is the distance from which the object can be seen. Thus, the brighter the object, the greater the distance it can be seen from. In a fly-through scenario the observer moves on a path (e.g., a trajectory on a density surface). This allows iToph to eliminate less important objects. These may be objects with a low visibility, or far-away objects with a high visibility.

Figure 1 shows an example of a universe and path. To guarantee that the observer will not see unimportant objects the length of the intersection of the visibility area and the path is used as a threshold. Specifically, iToph only retrieves the bold objects. The other objects are eliminated based on the length of the intersection of the path and the visibility area of the object.

3. PROBLEM DEFINITION

This section defines a visibility metric for non-equally sized objects, which we use to define the most visible objects $\mathcal{V}_\text{iTopN}$ from the set of visible objects $\mathcal{V}_P$. We start with some preliminary definitions. We write $P$ to denote a path, which consists of a sequence of observer positions. We write $\mathcal{V}_P$ for the visible objects from observer position $P$. Objects that become visible when an observer moves from position $P$ to position $P'$ are denoted by $\Delta_{P,P'}$. Equivalently, $\Delta_{P,P'}$ denotes the objects that cease to be visible when the observer moves from position $P$ to position $P'$. We write $|\mathcal{V}_P|$ to denote the number of objects that are visible from position $P$. We use $|p - P|$ for the distance between observer position $P$ and object $o$.

An object $o$ has a set of properties: $o,x$ (x coordinate of object position), $o,y$ (y coordinate of object position), and $o,s$ (size of object). The exact set of properties is not relevant for the purpose of this paper and can easily be extended. The visibility range defines the distance within which an object is visible. For the purpose of this paper the visibility range is proportional to the size of an object: $\mathcal{V}(o) = o.s \cdot c$. Thus, object $o$ will be visible in the circular or hyper-spherical area of size $\mathcal{V}(o)$ around the center of $o$.

The visibility area of an object is $V_A(o) = \{(x,y) \mid (x-o,x)^2 + (y-o,y)^2 \leq \mathcal{V}(o)^2\}$. We write $D(o) = \|P \cap V_A(o)\|$ for the length of the intersection of the path $P$ and visibility area $V_A(o)$. The visible objects from position $P$ with a relevance threshold of $W$ are defined as follows: $\mathcal{V}_P = \{o \mid o \in DB \land \|o - P\| \leq \mathcal{V}(o) \land D(o) \geq W\}$.

In order to rank the visible objects it is necessary to characterize the visibility of objects. We use the distance from the observer and the visibility range for this. The example in Figure 2 shows three differently sized objects. The circular areas are the VRs of the objects. The observer, marked

![Figure 1: Visual Data Exploration](image-url)
by a black cross, sees all three objects because the observer position lies within the visibility range of all objects.

**Definition 3.1. (Nearest Visible)** A nearest visible object from observer position $P$ is an object $o \in V_P$ such that $\forall o_j \in V_P: ||o_j - P|| > ||o - P||$.

This definition can be extended to the N-nearest visible objects. For example in Figure 2(a) object $o_2$, which is marked with a bold dashed line, is closer than object $o_1$ and $o_3$. This makes it the nearest visible object.

**Definition 3.2. (Largest Visible)** A largest visible object from position $P$ is an object $o \in V_P$ such that $\forall o_j \in V_P: VR(o) > VR(o_j)$.

This definition can be extended to the N largest visible objects. Contrary to the distance based nearest visible object definition, the VR is used as a selection parameter to compare objects. Figure 2(b) shows and compares the VRs of objects $o_1$, $o_2$, and $o_3$. It is clear that object $o_1$ has the largest VR and therefore can be seen from further away than the other objects.

To determine the most visible object we define a visibility metric.

**Definition 3.3. (Visibility Metric)** The visibility metric is a function $VM(o, P) = \frac{VR(o)}{d(o, P)}$, $o \in V_P$ there, $VR(o)$ is the visibility range of object $o$ and $||P - o||$ is the distance from the observer position $P$ to the center of the object. We require $||P - o|| > 0$ and $VR(o) > 0$.

The visibility metric function gives values in the range $[1; +\infty)$. A concrete value calculated by the $VM$ function is a visibility measure. The reason for the lower bound 1 is that we evaluate only visible objects, which means that the observer is inside the VR. The upper bound of the visibility measure goes to $+\infty$ because the observer can move infinitely close to an object. The intuition for this metric is that it combines two parameters, distance and VR of an object, in such a way that once the observer is on the visibility border the factor function is equal to 1. When moving closer to an object the measure becomes larger. For example, if the observer is on the borders of the visibility ranges of two differently sized objects they are treated as equally visible, i.e., the visibility measure is equal to 1. When the observer moves, the measure changes depending on the size of an object and the distance to it. The larger the value of the visibility measure is the more visible the object is.

**Definition 3.4. (N Most Visible)** Let $V_{TopN}^P \subseteq V_P$ and $|V_{TopN}^P| = N \leq |V_P|$. $V_{TopN}^P$ is a set of $N$ most visible objects iff $\forall o_1 \in V_{TopN}^P \forall o_j \notin V_{TopN}^P: VM(o_1, P) > VM(o_j, P)$.

The set $V_{TopN}^P$ contains a set of $N$ most visible objects. Note that there may be situations when several objects have the same visibility measure. This has to be taken into account when updating the set of most visible objects. To handle a new object $o_j$ with the same visibility measure as the least visible object $o_l$ in $V_{TopN}^P$ two approaches are possible. The first approach is to discard the new object if $VM(o_l, P) = VM(o_j, P)$. The consequence is that the $V_{TopN}^P$ is more stable. It means that incoming objects must have a strictly larger visibility measure to be inserted into $V_{TopN}^P$. The second approach would be to allow the old $o_l$ to be replaced by the new object $o_j$. The consequence is that in the worst case we end up updating all least visible objects in $V_{TopN}^P$.

**Theorem 3.1. (Distinctness of the Most Visible Object)** In the general case the most visible object differs from the nearest visible and the largest visible objects.

**Proof.** The proof is based on Figure 2(c). We have to show that (1) $VM_{most visible} > VM_{nearest}$, (2) $d_{most visible} > d_{nearest}$, and (3) $VR_{most visible} < VR_{largest}$. The geometrical interpretation is worked out in Figure 3. The nearest and the largest points produce two triangular areas. These areas marked in gray with different intensity. The overlap area is a triangle. Objects in this area are further away than the nearest visible objects and have a smaller VR than the largest visible object, but are still more visible in the sense of Definition 3.4. □

![Figure 2: Impact of Distance and VR](image)

![Figure 3: Bounds of d and VR for the Most Visible Object](image)
When the observer moves objects become visible and disappear as illustrated in Figure 4. The visibility range of the object is marked with the dashed line. We show the movement of the observer as an arrow. There are two objects that disappear (\(o^-\)) and one object that appears (\(o^+\)). One object remains stable (\(o\)).

**Definition 3.5. (\(\Delta^\pm\) Slices) Let the observer move from position \(P\) to \(P'\). The pair**

\[
(\Delta^+_{P,P'}, \Delta^-_{P,P'}) = (V^\text{TopN}_{P'} \setminus V^\text{TopN}_P, V^\text{TopN}_P \setminus V^\text{TopN}_{P'})
\]

**consists of the objects that enter and leave \(\text{TopN}\), respectively.**

The objects in \(\Delta^+\) and \(\Delta^-\) are the objects that become visible and invisible respectively when the observer makes a step.

**Theorem 3.2. (i\(\text{TopN}\) Construction from \(\Delta^\pm\)) Given a sequence of observer positions, \((P_0, \ldots, P_k) \in \mathcal{P}\) and the \(\Delta^\pm\) pairs for observer steps 0 \(\ldots\) \(i - 1\), \(i \leq k\) the \(\text{N most visible objects at observer position } P_i\) are:**

\[
V^\text{TopN}_{P_i} = \begin{cases} 
V^\text{TopN}_{P_{i-1}} \cup \Delta^+_{P_{i-1}, P_i} & i = 0 \\
\Delta^-_{P_{i-1}, P_i} & i > 0
\end{cases}
\]

Based on this formalization the main goals of the following sections are:

To find an efficient way to store and maintain \(V^\text{TopN}_P\).

To find an efficient way to compute the incremental slices \((\Delta^+_{P,P'}, \Delta^-_{P,P'})\) when the observer is moving. To show that using the previously stated goals and the formula in Theorem 3.2 we can compute \(V^\text{TopN}_P\) extremely fast.

## 4. CACHING TRACES

### 4.1 The VR-tree

To index visibility ranges we use the VR-tree, which optimizes the R-tree by modeling visibility ranges as *Minimal Bounding Squares* (MBS). An MBS approximates the visibility range of an object. Thus, we deal with three types of nodes. Internal nodes use MBRs to create a hierarchical structure. Leaf nodes use MBSs to approximate visibility ranges. Data nodes store the actual objects. By introducing leaf nodes with MBSs we increase the fanout of leaf-nodes. An MBS is constructed from object position and VR, while an MBR is constructed from two corner points.

Let \(D\) be the number of dimensions used, \(p\) be a D-dimensional point, and \(\text{Ptr}\) be the pointer to the child. An internal-node of a VR-tree holds up to \(\frac{b}{\text{Ptr} + |x| + |y| + 1}\) entries and leaf-nodes hold \(\frac{b}{\text{Ptr} + |x| + |y| + 1}\) entries. For example in 2D (3D) with a page size \(b = 8\) KB a leaf-node can hold 512 (409) entries and an internal node 409 (292) entries. We assume 4 bytes for a value and a pointer.

### 4.2 Volatile Access Structure

The volatile access structure (VAST) caches the traces of the most recently visited VR-tree nodes in memory. A VAST node consists of a number of entries. Each entry contains a value and a pointer to its child. The number of entries in a VAST node is dynamic, but is exactly the same as the number of entries in the corresponding VR-tree node.

Thus, VAST mirrors traces of the VR-tree. A trace is a sequence of pointers from the root to a leaf. Each pointer to a disk page in a trace is stored in the value field of VAST.

The overall goal is to query the VR-tree as little as possible by reusing previously accessed nodes that are stored in VAST. VAST is dynamic and entries are updated when the observer moves. Each entry in VAST node stores the distance \(d\) to the observer position \(P\), the angle \(\alpha\), the VR of the object, and the disk pointer to the VR-tree node it mirrors. The observer position is the center for the polar coordinate system and the angle is calculated with respect to the zero axis. For example, North corresponds to 0°. For internal nodes, the distance is calculated to the center of the MBR and the VR is the radius of the smallest enclosing circle.

The formula to calculate \(d\) and \(\alpha\) in 2D are

\[
\begin{align*}
\theta &= \sqrt{(P.x - o.x)^2 + (P.y - o.y)^2} \\
\alpha &= \arctan\left(\frac{P.x - o.x}{P.y - o.y}\right)
\end{align*}
\]

where \(P.x, P.y\) are the \(x\) and \(y\) coordinates for observer position \(P\). Formula (1) is used to compute distance, angle and VR for each node mirrored in VAST. Each node in VAST must be updated when the observer moves. Figure 5 shows an example of how VAST mirrors a subset of the VR-tree. In Figure 5(a) the VR-tree is shown. Two objects are visible from the observer position marked as cross (cf. Figure 5(b)). VAST creates an exact trace to the nodes shown in Figure 5(c). The intensity corresponds to the level in the VR-tree. In comparison to the visible objects VAST also caches all siblings of objects.

New nodes are added to VAST unless the observer hits an MBR that has not yet been expanded (cf. black arrows in Figure 5(c)). To generalize, there will be no incoming nodes in VAST unless the next observer position hits the MBR that is unseen. Otherwise the observer can move without issuing I/O operations.

To determine whether the observer is outside or inside of the visibility range of an object \(o\) in VAST we have to evaluate \(d - \text{VR}(o)\). The difference is the distance to the visibility range border. If \(d - \text{VR}(o) > 0\) the object was visible at an earlier point in time, but currently is not visible. If \(d - \text{VR}(o) \leq 0\) then the observer is inside the visibility range and the object is visible.

When the observer is moving, we must update VAST according to the distance and angle the observer is moving. In order to update VAST we descend from the root and update distance \(d\) and angle \(\alpha\) in each entry of VAST. The new distance \(d'\) and the new angle \(\alpha'\) are calculated as follows:

\[
\begin{align*}
d' &= \sqrt{(P'.x - P.x + d \cdot \cos \alpha)^2 + (P'.y - P.y + d \cdot \sin \alpha)^2} \\
\alpha' &= \arcsin\left(\frac{P'.y - P.y + d \cdot \sin \alpha}{d'}\right)
\end{align*}
\]

**Figure 4: A Moving Observer**
A size of a single entry of a leaf node or an internal node are the same. However an internal node of the VAST takes up to \( \frac{b}{D} \cdot (2 \cdot |P| + |d| + |a| + |VR|) \) · memory. This represents the number of VR-tree entries in a node multiplied by the size of an entry. The leaf-node memory consumption is calculated as follows: \( \frac{b}{D} \cdot (2 \cdot |P| + |d| + |a| + |VR|) \) · Thus, the modification of the R-tree is the same as issuing a query for the visible objects.

### 4.3 \( \Delta^\pm \) of All Visible Objects

The incremental algorithm that extracts \( \Delta^\pm \) for a unlimited number of visible objects works as follows:

1. Start the lookup at the root of VAST for the next observer position. Iterate over all entries and update distances and angles.

2. If an MBR does not intersect the current observer position then the MBR and the whole subtree are pruned. (Any VR that will appear in the hierarchy below is also invisible). Pruned leaf-nodes are added to \( \Delta^- \). No I/O operations are issued on the VR-tree.

3. If an MBR becomes visible then load the child from the VR-tree and mirror it in VAST. Note, that some child nodes might be visible as well. If a child is newly visible it is added to VAST and \( \Delta^+ \).

4. If the observer neither enters or leaves an MBR then descend to the child entries and repeat the algorithm recursively.

To summarize, the algorithm utilizes what was previously accessed and allows easy identification of leaving and incoming objects. I/O operations are issued only when a new subtree is read. The algorithm behaves well if the number of visible objects is large in comparison with the number of incoming and leaving objects. The structure keeps all visible objects and dynamically grows and shrinks. When the observer moves with steps that are large enough to leave all previously visible objects behind and a large set of new incoming objects is added, then the VAST structure is basically destroyed and re-created. Even in this case there is no overhead in I/O operations. The cost to recreate VAST is the same as issuing a query for the visible objects.

### 4.4 \( \Delta^\pm \) of TopN Objects

#### 4.4.1 VAST Extension For \( \times \text{TopN} \)

In order to retrieve the N most visible objects, the first extension of VAST is the addition of a buffer, TopN, of size N. The purpose of the buffer is to maintain a list of pointers to the most visible objects. Figure 6 shows an example of the enhanced VAST structure. The VAST structure is created at run time when the observer starts a data exploration. The TopN buffer is restricted in size and is sorted based on the visibility metric (\( V, M \)). The buffer entries point to VAST. The TopN buffer contains information to determine newly added pointers. More precisely, each pointer is associated with a one bit value that is set to 1 if the pointer is new and 0 otherwise.

#### 4.4.2 Function \( \Delta \times \text{TopN} \)

The function takes the current observer position \( P \), the next position \( P' \), the root node of the VAST structure, \( N \) the maximum number of top visible objects at the next observer position \( P' \), and the \( \times \text{TopN} \) buffer itself. The output of the function is the pair of \( \Delta s \). The pseudo-code of the function \( \Delta \times \text{TopN} \) is as follows:

```plaintext
Require: In: Obs \( P \), \( P' \); VAST root; int \( N \); \times \text{TopN} topn
Require: Out: (\( \Delta^+, \Delta^- \))
1: (\( \Delta^+, \Delta^- \)) = (\emptyset, \emptyset);
2: for all entries \( e \) in topn do
3: if becomesInvisible(\( P, P', e \)) then
4: \( \Delta^- \leftarrow \Delta^- \cup e \);
5: deleteFromTopEntries(\( N, topn, e \));
6: end if
7: end for
8: Start(\( P, P', N, topn \))
9: (\( \Delta^+, \Delta^- \)) = (\( \Delta^+, \Delta^- \)) \cup vastUpdate(\( P, P', \text{root}, N, topn \));
```

Figure 5: Initialization of the VAST

Figure 6: Enhanced VAST
The function scans the N most visible objects to determine the objects that become invisible. If an entry disappears then the object is added to $\Delta^-$ and the pointer is deleted from TopN. The next step is to prepare the remaining entries for a next observer position $P'$. Function Sort sorts pointers based on the visibility measure in the next observer position, but does not update entries. This is required because the new objects might be mirrored to VAST. When inserting into TopN buffer a pointer position among the other pointers is measured based on the visibility measure. Thus, it prevents from the full scan of TopN buffer. All pointers are marked as old pointers.

Function vastUpdate updates the VAST structure. The function updates distance and the angle from the current $P$ to the next observer position $P'$. Formula (2) is used to update distance $d$ and angle $\alpha$ for each entry. Newly visible objects are returned and a trace is added to VAST. $\Delta^-$ returns objects that were removed from the TopN buffer.

Many of these objects may remain visible and thus stay in VAST, but they are no longer among the top N most visible objects. The invisible nodes are pruned from VAST.

Function vastInsert updates the VAST structure. The function updates distance and the angle from the current $P$ to the next observer position $P'$. Formula (2) is used to update distance $d$ and angle $\alpha$ for each entry. Newly visible objects are returned and a trace is added to VAST. $\Delta^-$ returns objects that were removed from the TopN buffer. Many of these objects may remain visible and thus stay in VAST, but they are no longer among the top N most visible objects. The invisible nodes are pruned from VAST.

Insert Into VAST

Mirroring traces is the key function and is implemented in the VAST Insert function. Function VAST Insert is called with the next observer position and the node the insertion shall start from. A pointer to the representative node in the VR-tree is passed as parameter $r$. First, the VR-tree node is loaded from disk using $r$. Each entry of the VR-tree node is mirrored in to VAST. Distance $d$, angle $\alpha$, and VR of the MBR or MBS are calculated. Next, each disk pointer of VR-tree is assigned to VAST entries. The disk pointer enables us to directly access the relevant VR-tree node and load the missing subtree from the disk when it becomes visible. To decide which subtree to follow we check if the $d \leq VR$. If so, then the object is visible otherwise it is not visible. The child of an invisible entry is empty.

The pseudo-code shows the implementation of the insertion function vastInsert.

Require: In: $\text{Obs} P, P'$; $\text{VRTreePtr}$ $r$; VAST node; int $N$; TopN

1: $\text{VASTEntry} e$
2: $(\text{new, expired}) \leftarrow (\emptyset, \emptyset)$
3: $r\text{-node} \leftarrow \text{VRTreeNode}(r)$
4: for all entries $re \in r\text{-node}$ do
5: $e \leftarrow \text{mirrorEntry}(re, P')$
6: $VM \leftarrow \text{addEntry}(node, e)$
7: if $\text{isLeaf}(node)$ then
8: $(\text{new, expired}) \leftarrow (\text{new, expired}) \cup \text{vastInsert}(P, P',$
9: $\text{VRTreePtr}(r), \text{newChild}(e), N, \text{topn})$
10: end if
11: if $VM \geq \min(\text{topn}) \wedge \text{inTopN}(topn, e) \wedge \text{isLeaf}(node) \wedge D(e) \geq W$ then
12: $(\text{new, expired}) \leftarrow (\text{new, expired}) \cup \text{addToTopN}(topn, e)$
13: end if
14: if $\text{hasNoVisibleEntries}(node)$ then
15: deleteNode(node)
16: $new \leftarrow \emptyset$
17: end if

The TopN buffer is updated if the incoming node is a leaf node and is more visible than the entry with the least visibility measure in the TopN buffer. The condition $D(e) \geq W$ ensures that the leaf node entry has a visibility above threshold $W$. No objects with a lower visibility will be added to VAST and TopN, respectively.

If the function addToTopN is called then the new entry is evaluated against all entries in the TopN buffer. The function is implemented as a linear scan function. It scans the TopN buffer and evaluates $VM$ for all old and new pointers. If the entry is added to TopN then it is added to the set of the new objects and the pointer is marked as a new pointer. It might happen that the buffer is full and the insertion requires the removal of the least visible object from the buffer. If the least visible object in the TopN is a new pointer then it is removed from the new set. The new incoming entry is added to new. If the least visible entry is an old pointer then we remove the entry from the TopN and add it to expired.

After all entries have been mirrored vastIns checks if the node has no visible entries by calling the function hasNoVisibleEntries. It might happen that an insertion does not lead to a successful visible child. These nodes are deleted and the recursion is backtracked with the empty set of the new objects.

5. EXPERIMENTS

We have implemented VAST using the Generalized Search Tree package, GIST [5]. Throughout the experiments we use a two dimensional VR-tree implementation.

Level-Wise Caching with LRU:

In order to compare $\sharp\text{TopN}$ to current technology we implement Level-Wise with LRU (LW-LRU) caching. We expect that given a fair amount of main memory and an advanced caching policy LW-LRU might be competitive with the TopN most visible object extraction using the VAST. Using plain Least Recently Used (LRU) caching fails because when the buffer is filled the next incoming node overwrites the least recently used node that can be located high in the R-tree hierarchy. The LRU cache starts to degenerate when nodes that are high in the hierarchy are being flushed. Therefore, caching top level nodes of the R-tree is essential.

To ensure that top level nodes are not flushed out too soon we allow a number of top level nodes to be pinned. In general the cache is divided into two parts: LW, which caches according to the level, and LRU, which caches according to the access frequency. We expect that such a generalized LRU with a number of pinned nodes (LW) is competitive to VAST. The LW cache keeps track of partial traces to the visible objects and is defined as follows. For each accessed node in the R-tree, a node is inserted into the LW buffer based on the level in the R-tree. Nodes are grouped according the level and inserted strictly according to the level. If the node is already in LW buffer no I/O is needed. If the node is not in LW buffer and the buffer is full, and if the accessed node is higher than the last LW node then the node replaces the least recently used node of the level below. If the LW buffer is full and an accessed node is at a lower level than the last node stored in the cache then the node is cached in LRU.

5.1 Click-stream Dataset

The first set of experiments is conducted on a click-stream dataset. The click-stream data is a log of web-page accesses at the department of computer science at Aalborg University. The log contains 1 million observations over a three month period. We map attributes host and time to dimensions X and Y. The size of a document is mapped to the VR attribute. The intention is that priority is given to the largest downloads.
The experiment compares TopN for different N, and the retrieval of all visible objects using the R-tree with LW-LRU. We compare the average number of I/O operations when the step size is changing following the same path. We create two sets of observer movement paths through the data. Each route has paths of equal step sizes. We consider 0.02%, 0.03%, 0.07%, 0.1%, 0.14% of the universe size. This a percentage is based on the data density of the visible objects along the path. For example, the average VR of objects in the universe 0.7%. The same set of paths is used in the comparison of all methods.

In the first experiment, the observer path goes through the most dense area. This pushes all methods to an equally bad situation. We expect that given a certain amount of memory the LW-LRU cache can behave as well as TopN.

In Figure 7 we write LW-2 and LW-3 for LW-LRU caching where nodes from the 2 and 3 top most levels are permitted in LW. We use Top-N as the non restricted TopN algorithm. Which permits all visible objects. The result is produced by the algorithm of incremental slices of all visible objects (cf. Section 4.3). Top-100 and Top-10 restrict the N most visible objects to 100 and 10, respectively. They are produced with the \( \Delta ITopN \) algorithm. During the experiment the number of VAST nodes ranges in [800, 1300]. LW caching uses a constant amount of space allocated for node caching that is three times larger.

With a larger step size LW degenerates faster (cf. Figure 7). The reason is that LW gets filled with nodes from high-up in the R-tree. On the contrary, VAST prunes unused traces and leaves space for new objects. Another interesting point is that the non-restricted algorithm of all visible objects (Top-N) is an upper bound for \( \Delta ITopN \). It means that if we choose N large enough to accommodate all visible objects, \( \Delta ITopN \) always ends up with the same number of I/O operations as the non restricted incremental extraction.

The second experiment is done for the observer moving from a less dense area to a dense area. We choose LW-3 with a cache size of 200, 400, and 600 nodes, respectively. The observer uses the same step sizes as in the previous experiment. Figure 8 reports the average number of I/O operations. LW-3 cache of 600 nodes (LW-3/600) is a lower bound for the R-tree because it caches all traces. Allowing the number of LW nodes to be larger than 600 will lead to almost the same number of I/O. Essentially, only newly visible objects require an I/O because the other objects fit into the LW cache. We have confirmed this by choosing a different path where the observer revisits a place. As expected the number of I/O dropped to zero because the previously accessed traces are cached in LW.

![Figure 7: Unrestricted LW Cache Versus TopN](image)

Figure 7: Unrestricted LW Cache Versus TopN

![Figure 8: LW Cache With 200, 400, and 600 nodes Versus TopN](image)

Figure 8: LW Cache With 200, 400, and 600 nodes Versus TopN

Top-N, which also retrieves all visible objects, is an upper bound for VAST. In this particular experiment LW-3/600 uses 6 times more memory than VAST. Experiments with different datasets confirm this difference: to get an optimal performance with LW one needs three to four times as much memory as with TopN. Note that the best performance of LW is significantly worse than the performance of VAST. Notice that in Figure 8 when observer moves with step-size 0.1. All methods reflect the average I/O reduction. It is because a certain steps yield the path where only a few objects are visible therefore decreasing the number of incoming objects.

5.2 Synthetic Data

The next experiment is conducted on a torus like dataset (cf. Figure 9). The torus consist of 10,000 normally distributed objects scattered in a region of size 100x100. The objects near the center have been removed. The inner border of the torus is more dense than the outer. The visibility ranges vary from 0.1%–5% of the universe size and are normally distributed. The dataset allows us to investigate how a changing density impacts the I/O behavior.

In Figure 9, we plot the data points and the path the observer follows. The path is constructed of 100 path points and the observer moves through the space from (0,0) to (100,100).

The comparison of the result-set size is shown in Figure 10. The upper line shows the size of results extracted using the R-tree solution. The lower line shows the number of results with a non-restricted incremental extraction of visible objects. Clearly, for \( \Delta ITopN \), the number of extracted objects is much less than the number of visible objects which shows the effectiveness of an incremental extraction.

![Figure 9: Rapid Change in Density of Data When Moving](image)

Figure 9: Rapid Change in Density of Data When Moving
Figure 10: Moving Through Dense and Sparse Areas

Based on these experiments we draw the following conclusions:

- In order to get the best performance with R-trees and LW caching one has to allocate three times as much memory as required for VAST. If LW-LRU and VAST are using the same amount of main memory the R-tree uses substantially more I/O operations because LW cache is not utilized well.

- The best performance of LW-LRU is and order of magnitude worse than the worst performance of \textit{iTopN}.

- \textit{iTopN} is less expensive in terms of I/O operations than a incremental extraction of all visible objects. Experiments confirm that the average number of I/O operations depends on the object visibility, density and the step-sizes.

6. CONCLUSIONS AND FUTURE WORK

In this paper we introduce \textit{iTopN}, an incremental approach to extract the N most visible objects. We introduce \(\Delta^2\) slices, which report incoming and leaving objects for each observer position. We define a visibility metric, which identifies the objects based on the two parameters: distance and the size of the object. We explain how the VR-tree together with VAST extracts all visible objects incrementally.

We confirm our goals and expectations with experiments. In order to compare the R-tree with VAST we propose a Level-Wise (LW-LRU) caching policy for R-trees.

Further enhancements of VAST might include the optimization of VAST based on the speed the observer is moving, a priori knowledge of the path, and visible object density estimation. The current VAST implementation does not allow nodes to expire after they become invisible. To do more efficient memory maintenance, function \(\Delta \text{iTopN}\) can expire nodes and prune all invisible sub-trees. If the size of the memory that is allocated to VAST is restricted the expiration policy should further enhance the performance of VAST.

7. REFERENCES


