Multidimensional Data and Modelling
A general approach, can deal with inseparable objects
Automatic, uses as partitions planes defined by the scene polygons
Method has two steps:
  - building of the tree independently of viewpoint
  - traversing the tree from a given viewpoint to get visibility ordering
BSP tree is a general solution, but with its own problems
  - Tree size
  - Tree accuracy
Indexing hierarchies (BSP-trees)

A set of polygons

\{1, 2, 3, 4, 5, 6\}

The tree
Indexing hierarchies (BSP-trees)

- (recursive)

Select one polygon and partition the space and the polygons
Indexing hierarchies (BSP-trees)

- (recursive)

Recursively partition each sub-tree until all polygons are used up
The tree can also be built incrementally:
- start with a set of polygons and an empty tree
- insert the polygons into the tree one at a time
- insertion of a polygon is done by comparing it against the plane at each node and propagating it to the right side, splitting if necessary
- when the polygon reaches an empty cell, make a node with its supporting plane
Indexing hierarchies (BSP-trees)

Each node corresponds to a region of space
the root is the whole of $\mathbb{R}^n$
the leaves are homogeneous regions
Indexing hierarchies (BSP-trees)
Indexing hierarchies (BSP-trees)
Indexing hierarchies (BSP-trees)

- BSP tree for dynamic scenes:
- when an object moves the planes that represent it must be removed and re-inserted
- some systems only insert static geometry into the BSP tree
- otherwise must deal with merging and fixing the BSP cells
- choice of plane used to split is critical
- BSP-trees are hard to maintain for dynamic scenes
Indexing hierarchies (hB-tree)

- A multiattribute index structure corresponds to the hB-tree.
- It is derived from the K-D-B-tree but has additional desirable properties.
- The hB-tree (holey brick tree – kiauros plytos medis) is similar to k-d-B-tree, except that splitting of the node is done based on *multiple attributes*, with external *enclosing regions* and several cavities called *extracted regions*.
- The hB-tree internal node search and growth processes are precisely analogous to the corresponding processes in B-trees.
Indexing hierarchies (hB-tree)

- We distinguish between data nodes that are pages which contain the records of the database and index nodes that contain k-d trees.
- The data nodes are the leaves of the hB-tree. The index nodes are the internal nodes of the hB-tree.
- The hB-tree grows from the leaves and has all leaves at the same level, just as a B-tree does.
- Specific node structure: the k-d tree is for internal structure of the index node - within hB-tree index nodes is to organize information about lower levels of the hB-tree. The k-d tree also assists in organizing data nodes of hB-trees.
A holey brick is represented via k-d tree. A holey brick is a brick from which a smaller brick has been removed. Two leaves of the k-d tree are required to reference the holey brick region denoted by B.
Indexing hierarchies (hB-tree)

- Bricks share boundaries. For a k-d tree, a boundary is typically checked only once.
Indexing hierarchies (hB-tree)

- Searching using the hB-tree:
  - Exact Match Queries: one follows a unique path from the root of the hB-tree down the hB-tree to a data page. The number of hB-tree nodes accessed is the height of the hB-tree.
  - Range Queries. A range query specifies a range of values for one or more of the attributes. If a comparison value in a k-d-tree node is greater than all the same attribute’s values in the search range, one goes to the left. If it is smaller, one goes to the right. If a comparison value is in the middle of a search range, one follows both the left and right branches. Several hB-tree nodes at each level may be accessed.
  - Region Data: Finding all the regions D contained in a given region R is a range query.
Indexing hierarchies (hB-tree)

- Data node splitting:

```
\[\begin{array}{|c|c|}
\hline
x_1 & B \\
\hline
C & \hline
\end{array}\]

A

before split

\[\begin{array}{|c|c|}
\hline
x_1 & x_2 \\
\hline
B & D \\
\hline
C & \hline
\end{array}\]

A

after split

\[\begin{array}{|c|c|}
\hline
x_1 & y_1 \\
\hline
C & B \\
\hline
A & \hline
\end{array}\]

\[\begin{array}{|c|c|}
\hline
x_1 & y_1 \\
\hline
C & x_2 \\
\hline
B & D \\
\hline
A & \hline
\end{array}\]

A

The usual case for a data node split. The data node B is a brick, and the split is along the x attribute at \(x_p\) forming two bricks. One brick remains at disk address B while a new brick is allocated at D. With at most k attribute values (in a k-dimensional space) determining a “closed corner” as in Figure 6, we can guarantee at least a 2:l split in the data points for any point distribution. That is, the data points can be divided so that no more than \(5\) of them are in one node and no less than \(\frac{1}{2k}\) of them are in the other. We prove this in Appendix 1. We can guarantee minimum data node utilization of at least \(t\). Note that we usually do much better than 2:l.

Indexing hierarchies (hB-tree)

- Index node splitting:

Fig. 9. The same example as in Figure 8, where the k-d tree of an index node is too skewed to split at the root. We find a subtree which has between $\frac{1}{3}$ and $\frac{2}{3}$ of the nodes and extract that subtree. In this case, then, is that in one resulting node utilization begins at $\frac{2}{3}$ and in the other at $\frac{1}{3}$. When an index node is split, we must post information in the parent to distinguish the extracted tree from the enclosing tree. The extracted tree represents a k-dimensional brick. Thus, at worst, we need to post the $2k$ boundaries to the parent. However, by using a subset of the nodes on the path from the root of the extracted tree to the root of the enclosing tree, we can do better most of the time. Also, in this way, the algorithm for posting information is simpler. However, to gain these advantages, we must occasionally post one additional node, so that in the worst case we post $2k + 1$ k-d tree nodes as our hB-tree index term. We give the details of this algorithm in Section 6.

Determining storage utilization in index organizations subject to uneven splitting was a problem faced in Digital B-trees [8]. The analysis done there used the uniform growth assumption and involved computing recurrence relations.

Indexing hierarchies (hB-tree)

- A k-d tree is used as the structure within nodes for very efficient searching.
- Node splitting requires that this k-d tree be split.
- This produces nodes which no longer represent brick-like regions in k-space, but that can be characterized as holey bricks, bricks in which subregions have been extracted.
- The intranode processes are unique.
- Results guarantee hB-tree users decent storage utilization, reasonable size index terms, and good search and insert performance.
Indexing by subspaces (hB-tree)

hB-tree
The Isd tree: spatial access to multidimensional point and non-point objects

Local Split Decision tree (LSD tree) - data structure supporting efficient spatial access to geometric objects.

It performs well for all reasonable data distributions, cover quotients (which measure the overlapping of the data objects), and bucket capacities, and that it maintains multidimensional points as well as arbitrary geometric objects.
Indexing hierarchies (LSD-tree)

- These properties demonstrated by an extensive performance, evaluation make the LSD tree extremely suitable for the implementation of spatial access paths in geometric databases.
- The paging algorithm for the binary tree directory is interesting in its own right because a practical solution for the problem of how to page a (multidimensional) binary tree without access path degeneration is presented.
Like most structures, the new structure partitions data space into pairwise disjoint cells with associated buckets of fixed size.

The free choice of split positions is the basis of the graceful adaptation to arbitrary skew object distributions.

Since a new split position can be chosen locally optimal, i.e. optimal with respect only to the cell to be split and independent from other existing cell boundaries, the new structure is called Local Split Decision tree (LSD tree, for short)
Indexing by subspaces (LSD-tree)

LSD tree associated with the data space partition
Indexing hierarchies (LSD-tree)

Besides the advantages of the LSD tree directory there are some drawbacks typical for multidimensional binary tree structures:

- A multidimensional binary tree may become unbalanced, i.e. may contain long paths with almost no branches, and
- no suitable method for paging a multidimensional binary tree is known.

To overcome these problems we introduce a paging algorithm with **external balancing property**: the number of external pages which are traversed on any two paths from the directory root to a bucket differs by at most 1.
Indexing hierarchies (LSD-tree)

- When geometric objects are inserted into an initially empty LSD tree the directory grows up to a size when it cannot be kept in the dedicated part of the main memory any longer.
- Then the paging algorithm determines a subtree to be paged on secondary storage such that the external balancing property is preserved.
- If the subtree consists of \( n_1 \) nodes, the main memory is then able to receive additional \( n_2 \) nodes until a further invocation of the paging algorithm must take place.
Overall structure of LSD-tree
Indexing hierarchies (LSD-tree)

- The search for bucket $b$ which will receive new object is guided by directory as in k-d-trees.
- If $b$ does not overflow, the insertion is finished, otherwise bucket split algorithm creates two new buckets $b_1$ and $b_2$ from $b$ according to a split strategy.
- In case of a bucket split the pointer in directory referencing $b$ is changed to a pointer referencing a new directory node $q$, representing the split decision concerning $b$, i.e. the new node $q$ is inserted into the directory by calling the directory insertion algorithm.
- The new node $q$ points to new buckets $b_1$ and $b_2$. 
Effect of a bucket split of LSD-tree

![Diagram showing the effect of a bucket split of LSD-tree](image)

- **Directory root**
- **Split of b**
- **Directory node or data bucket**
- **New buckets** $b_1$ and $b_u$
Indexing hierarchies (LSD-tree)

The paging algorithm is called - when after an insertion of an additional node the size of internal prefix tree $T_i$ reaches the maximal possible number $n_i$ of internal nodes.

The algorithm searches for a subtree $T_s$ in $T_i$ such that paging $T_s$ preserves the external balancing property.

This property is preserved if $T_s$ is a paging candidate, i.e. $T_s$ fullfills the following properties:

- Any path from root of $T_s$ down to a bucket contains minimal number of external directory pages (of all paths in directory $T$).
- The height of $T_s$ is at most $h_T$
Directory before and after paging

In directory traversal, we encounter the following operations:

1. Traversing from the root of the tree to a node on path $P$.
2. Updating the height of a node, $h(v)$.
3. Querying properties of the directory, such as the number of nodes.

These operations are performed to ensure that the directory is efficiently searched and managed. The figure illustrates the directory structure before and after paging, showing how the directory changes during these operations.