Efficient Searching for Multi–dimensional Data Made Simple (Extended Abstract)

Enrico Nardelli[1], Maurizio Talamo[2], and Paola Vocca[3]

[1] Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, Via Vetoio, Coppito I-67010 L'Aquila, Italy, nardelli@univaq.it
[2] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, I–00198 Rome, Italy, talamo@dis.uniroma1.it
[3] Dipartimento di Matematica Università di Roma "Tor Vergata", Via della Ricerca Scientifica, I–00133 Rome, Italy, vocca@mat.uniroma2.it

Abstract. We introduce an innovative decomposition technique which reduces a multi–dimensional searching problem to a sequence of one–dimensional problems, each one easily manageable in optimal time×space complexity using traditional searching strategies. The reduction has no additional storage requirement and the time complexity to reconstruct the result of the original multi–dimensional query is linear in the dimension.
More precisely, we show how to preprocess a set of $S \subseteq \mathbb{N}^d$ of *multi–dimensional* objects into a data structure requiring $O(m \log n)$ space, where $m = |S|$ and n is the maximum number of different values for each coordinate. The obtained data structure is *implicit*, i.e. does not use pointers, and is able to answer the *exact match* query in $7(d-1)$ steps. Additionally, the model of computation required for querying the data structure is very simple; the only arithmetic operation needed is the addition and no shift operation is used.
The technique introduced, overcoming the multi–dimensional bottleneck, can be also applied to non traditional models of computation as external memory, distributed, and hierarchical environments. Additionally, we will show how the proposed technique permits the effective realizability of the well known perfect hashing techniques on real data.
The algorithms for building the data structure are easy to implement and run in polynomial time.

1 Introduction

The efficient representation of multi–dimensional points set plays a central role in many large–scale computations, including, for instance, object management in distributed environments (CORBA, DCOM); object–oriented and deductive databases management [2,5,25,10,19], and spatial and temporal data manipulation [20,24]. All these applications manage very large amounts of multi–attribute data. Such data can be considered as points in a d–dimensional space. Hence, the key research issue, in order to provide "good" implementations of these applications, is the design of an efficient data structure for searching in the d–dimensional space. A fundamental search operation is the *exact match* query, that is, test the presence of a point in the multi–dimensional set

J. Nešetřil (Ed.): ESA'99, LNCS 1643, pp. 339–353, 1999.

when all its coordinates are specified. Another important operation is the *prefix–partial match* query which looks for a set of points, possibly empty, for whom only the first $k \leq d$ coordinates are specified.

We deal with the exact match query by using an innovative decomposition technique which reduces a multi–dimensional searching problem to a sequence of one–dimensional problems, each one easily manageable in optimal time$\times$space complexity using traditional searching strategies. The reduction requires no additional storage besides that one required for data and the time complexity to reconstruct the result of the original multi–dimensional query is linear in the dimension. The technique introduced, overcoming the multi–dimensional bottleneck, can be applied in more general contexts, such as distributed and hierarchical environments. Additionally, it can be positively used, jointly with perfect hashing techniques, when dealing with real data.

The technique is based on two main steps. In the first step, we reduce the d–dimensional searching problem to a sequence of d one–dimensional searching problems. In the second step, the multi–dimensional data is reconstructed using a set of $(d-1)$ 2–place functions. Each function is represented using a new data structure derived from a decomposition of the 2–place functions into a set of "sparse" 2–place functions easily representable. The decomposition technique of 2–place functions is an application of a more general technique introduced in [22] and successively refined in [23] for testing reachability in general directed graphs. The same technique has been successfully applied in [21] to the problem of implicitly representing a general graph.

The data structure we present has the following characteristics:

- *general and deterministic*: We represent any multi–dimensional point set and our space and time bounds are worst-case deterministic;
- *space and time efficient*: Exact match query requires $7(d-1)$ steps and prefix–partial match $7(k-1)+t$ steps, where t is the number of points reported, using $O(m \log n)$ space, where m is the size of the point set and n is the maximum number of values a coordinate can receive;
- *easy to implement*: The algorithms used to build the data structure, although somewhat tricky to analyze, are very simple and run in $O(n^3)$ time; no operations are needed for searching other than one–dimensional array accesses;
- *simple computation model*: The only arithmetic operation required for querying the data structure is the addition and no shift operation is used.

Due to its relevance, the multi–dimensional searching problem has been deeply investigated. In computational geometry and for spatial databases, the problem has been solved only for small values of the dimension [18,20] and the solutions proposed grow exponentially with d. The same problem has been studied for temporal databases [24]. In this case, we have empirical results, only, and the worst case is unbounded. In a general setting, there are two major techniques for implementing the multi–dimensional searching problem: trees and hashing. For the first, several data structure have been developed as d-dimensional version of data structures for the one–dimensional problem (e.g. B-trees [3], compacted tries [17], digital search trees [14]). In this case, even though the space complexity is optimal exact match queries require a logarithmic number of steps in the worst case. Hashing and perfect hashing techniques have the drawback that, for each search performed, it may be required the evaluation of computationally complex functions [8,7,6]. Hence, numerically robust implementations are required.

With our technique, each search only requires a constant number of table accesses, and addresses to be accessed are computed with only a constant number of additions.

Concerning the comparison of our technique with the less powerful computational models considered in the so–called word-RAM approach [9], namely the RISC model, the are two issues to be considered. First, our technique does not need to use a shift operation, which may require at least $\log m$ additions to be simulated, where m is the problem size. Second, the overall space needed for computations in the word-RAM model is $O(2^w)$ bits, where w is the word size, and for the model to be of interest this quantity has to be considerably larger than the problem size m, namely $2^w \gg m$ ([9], pag. 371). Contrast this with the overall space needed in our approach that, expressed in terms of m, can be written as $O(m \log^2 m)$.

The paper is structured as follows: In Section 2 we describe the representation of the multi–dimensional problem by means of a sequence of 2–place functions; In Section 3 we give some definitions and notations, and present some decomposition theorems; Using these theorems, in Section 4 we describe the data structure for representing a 2–place function and, hence, a multi–dimensional points set; then, in Section 5 we present some application of our technique. Finally, in Section 6 we outline some open problems and future research directions.

2 Problem Representation

In this section, we show first how to reduce a multi–dimensional problem to a set of one–dimensional problems, and then how to reconstruct the original problem.

Given $S \subseteq \mathbb{N}^d$, with $m = |S|$. Let $x = x_1, \ldots, x_d \in S$, then $n_i \doteq |\{x_i : x \in S\}|$. The reduction is defined by the following set of functions:

$$g_i : \mathbb{N} \longmapsto \{1, \ldots n_i\} \qquad 1 \leq i \leq d. \tag{1}$$

Each function g_i maps the values of a coordinate to a set of integers of bounded size. This mapping can be easily represented with data structures for one–dimensional searching, such as B-trees or perfect hashing tables. Without loss of generality, from now on we assume $S \subseteq U^d$, where $U = \{1, \ldots n\}$, being $n = max_i\{n_i\}$.

Hence, let $x = x_1, \ldots, x_d \in S \subseteq U^d$ be a generic key of S, where x_i denotes the value of the i-th coordinate. Let $a = a_1, a_2, \ldots, a_d$ be a value in U^d. We denote with $a(i)$ the subsequence of its first i coordinates, namely $a(i) = a_1, a_2, \ldots, a_i$, called a *partial value* or the *prefix* (*of length* i) of a. We write $b(j) \subset a(i)$ when $j < i$ and $b_k = a_k$, for $k = 1, 2, \ldots, j$. In the same way, we define the prefix for a key in S.

Let $S(a(i))$ be the subset of S containing all keys that are coincident on the prefix $a(i)$. Note that $|S(a(d-1))| \leq n$ and $|S(a(d))| \leq 1$. For any $a(i)$ such that $|S(a(i))| > 1$ and $\nexists b(j) \subset a(i)$ such that $S \supseteq S(b(j)) \supset S(a(i))$ we say that $a(i)$ is the *maximal shortest common prefix* of $S(a(i))$ *with respect to* S. We assume that it does not exist a maximal shortest common prefix $a(i)$ such that $S(a(i)) = S$, since otherwise we can consider a reduced dimension universe, by simply deleting the maximal shortest common prefix from every key.

The representation mechanism we use for keys is based on a suitable coding of subsets of keys with common prefixes of increasing length, starting from the maximal shortest common prefixes. We denote with f_i a 2-place function such that $f_i : U^i \times U \mapsto S$. We code keys using these functions in an incremental way.

Given a set T of keys, we denote with $s_l^T, 1 \leq l \leq k_T$, the l-th key in a fixed, but arbitrarily chosen, total ordering of the k_T keys in T. The choice of the order is immaterial: we use it only to make the description clearer.

Let us now assume $a(i)$ is a maximal shortest common prefix with respect to S. For reasons that will be clearer in the following, we only take into account maximal shortest common prefixes longer than 1. We then represent $S(a(i)), i > 1$, with the following technique.

First we represent the $i-1$ smallest elements in $S(a(i))$ as it follows:

$$\begin{cases} f_1(a_1, a_2) = s_1^{S(a(i))} \\ \dots \\ f_{i-1}(a_1 \dots a_{i-1}, a_i) = s_{i-1}^{S(a(i))} \end{cases}$$

Now, if $k_{S(a(i))} \leq i-1$, we have represented all elements in $S(a(i))$ and we are done. Otherwise we still have to represent the $k_{S(a(i))} - (i-1)$ remaining elements in $S' = S(a(i)) \setminus \bigcup_{l=1}^{i-1} s_l^{S(a(i))}$.

All keys in S' can then be partitioned in subsets, possibly just one, each containing keys with a common prefix $a(i+j) \supset a(i)$, and such that, for each subset S'_r, $a(i+j_r)$ is the maximal shortest common prefix of $S'_r = S(a(i+j_r)) \cap S'$ with respect to S'.

We now represent the $k_{S'_r}$ keys in S'_r by recursively applying the same approach.

Namely, we first represent the j_r smallest keys in S'_r as it follows:

$$\begin{cases} f_i(a_1 \dots a_i, a_{i+1}) = s_1^{S'_r} \\ \dots \\ f_{i+j_r-1}(a_1 \dots a_{i+j_r-1}, a_{i+j_r}) = s_{j_r}^{S'_r} \end{cases}$$

Now, if $k_{S'_r} \leq j_r - 1$, we have represented all elements in S'_r and we are done. Otherwise, we still have to represent the $k_{S'_r} - (i+j_r-1)$ remaining elements in $S''_r = S'_r \setminus \bigcup_{l=1}^{j_r} s_l^{S'_r}$.

All keys in S''_r can then be partitioned in subsets, possibly just one, each containing keys with a common prefix $a(i+j_r+h) \supset a(i+j_r)$, and such that, for each subset $S''_{r,q}$, $a(i+j_r+h_{r,q})$ is the maximal shortest common prefix of $S''_{r,q} = S(a(i+j_r+h_{r,q})) \cap S''_r$ with respect to S''_r. And now the representation process goes on recursively.

We now show an example of the application of the definitions introduced above.

Example 1. Assume $d = 6$ and $n = 9$. Consider a set $S = \{233121, 233133, 233135, 233146, 234566, 234577, 234621, 234622, 234623, 343456\}$. Then there are only two maximal shortest common prefixes with respect to S, namely 23 of length 2 and 343456 of length 6.

We then set $f_1(2,3) = 233121$ and $f_1(3,4) = 343456$: since $k_{S(23)} \not\leq 2-1$ while $k_{S(343456)} \leq 6-1$ then $S(343456)$ has been completely represented, while for keys remaining in $S' = S(23) \setminus \{233121\}$ we have to recursively apply the same technique.

The maximal shortest common prefixes in $S' = \{233133, 233135, 233146, 234566, 234577, 234621, 234622, 234623\}$ are 2331 of length $2+2$ and 234 of length $2+1$. It is $S'_1 = S(2331) \cap S' = \{233133, 233135, 233146\}$ and $S'_2 = S(234) \cap S' = \{234566, 234577, 234621, 234622, 234623\}$.

We then set $f_2(23, 3) = 233133$ and $f_3(233, 1) = 233135$; we also set $f_2(23, 4) = 234566$. Since $k_{S'_1} \not\leq 2$ and $k_{S'_2} \not\leq 1$ then both for keys remaining in $S''_1 = S'_1 \setminus \{233133, 233135\}$ and for those in $S''_2 = S'_2 \setminus \{234566\}$ we have to recursively apply the same technique. We obtain the following sets: $f_4(2331, 4) = 233146$, $f_3(234, 5) = 234577$, $f_3(234, 6) = 234621$, $f_4(2346, 2) = 234622$, and $f_5(23462, 3) = 234623$.

Given a d–dimensional set of points $S \subseteq U^d$ a point $x = a_1, a_2, \ldots, a_d$ can be searched by incrementally evaluating the 2–place functions f_i. At each step i, with $i = \{1, \ldots, d-1\}$, two cases are possible: $f_i(a_1 \ldots a_i, a_{i+1}) = x$ and we are done. Otherwise, the search continues with the evaluation of f_{i+1}. It is trivial to verify that the search ends reporting x if and only if $x \in S$. In the next section we show how to efficiently represent 2–place functions so that the above search strategy can be executed in a constant number of steps.

3 2–place Functions Representation

In order to state the main result of this section we need to recall some definitions and give new notations.

3.1 Definitions

A *bipartite graph* $G = (A \cup B, E)$ is a graph with $A \cap B = \emptyset$ and edge set $E \subseteq A \times B$.

Given a 2–place function $f : A \times B \longmapsto \mathbb{N}$, a unique labeled bipartite graph $G = (A \cup B, E)$ can be built, such that the label of $(x, y) \in E$ is equal to z if and only if $x \in A$, $y \in B$, and $f(x, y) = z \in Z$. Hence, the representation of a 2–place function is equivalent to test adjacency in the bipartite graph and lookup the label associated to the edge, if it exists. For ease of exposition, in the following, we will deal with labeled bipartite graphs instead of 2–place functions. Moreover, from now on, n_A and n_B denote the number of vertices in A and B, respectively, and m is the number of edges of the bipartite graph.

Given a bipartite graph $G = (A \cup B, E)$, $x \in A \cup B$ is *adjacent* to $y \in A \cup B$ if $(x, y) \in E$. Given a vertex x, the set of its adjacent vertices is denoted by $\alpha(x)$; $\delta(x) \doteq |\alpha(x)|$ is the *degree* x. The notation is extended to a set S of vertices as $\alpha(S) \doteq \cup_{x \in S} \alpha(x)$ and $\delta(S) \doteq \sum_{x \in S} \delta(x)$. The maximum degree among vertices in S is denoted by Δ_S. In particular, Δ_A and Δ_B denote the maximum degree among vertices in A and B, respectively. A bipartite graph is *regular* if all vertices have the same degree $\Delta = \Delta_A = \Delta_B$. A bipartite graph $G = (A \cup B, E)$ is *bi-regular* if all vertices in A have the same degree Δ_A and all vertices in B have the same degree Δ_B.

Given a set of vertices $S \in A$ or $S \in B$, $\delta_S(x) \doteq |\alpha(x) \cap S|$ denotes the number of vertices in S adjacent to x. Furthermore, $\alpha_j(S) \doteq \{x \in \alpha(S) : \delta_S(x) = j\}$ denotes the set of vertices in $\alpha(S)$ incident to S with exactly j edges. Given a set of vertices $S \in A \cup B$, the *sub-bipartite induced by* S is the sub-bipartite $G' \doteq (S, E_S)$, with $E_S = E \cap (S \times S)$.

A h-*cluster* S is a set of vertices, either in A or in B, s.t. $\delta_S(x) \leq h$, $x \in \alpha(S)$. A 1-cluster is simply called *cluster*.

3.2 Partitioning into h-Clusters

We present an algorithm which, given a bipartite graph $G = (A \cup B, E)$, computes a h-cluster $C \subset A$, with $h = \lceil \log n_B \rceil$; hence, the sub-bipartite induced by $C \cup B$ has the property $\Delta_{\alpha(C)} \leq h$. Of course, this can be done trivially if C consists of at most h vertices. Somewhat surprisingly, it turns out that a clever selection of vertices of the h-cluster, we can find a h-cluster of $\Omega\left(\frac{n_A}{\Delta_B}\right)$ vertices, hence a significant fraction of all vertices in A.

The idea behind the algorithm derives from the following observation: when we add a new vertex x to the h-cluster, then for each vertex y in $\alpha(x)$, its degree $\delta_C(y)$ with respect to C increases by one. A trivial approach would be to just check that for each vertex $y \in \alpha(C) \cap \alpha(x)$, $\delta_C(y) \leq h - 1$ holds; this guarantees $\Delta_{\alpha(C)} \leq h$ after the insertion. Unfortunately, on the long run this strategy does not work. A smarter strategy must look forward, to guarantee that not only the current choice is correct, but that it does not restrict too much successive choices. A new vertex x is added to the cluster in h successive steps, at each step j observing how x increases the number $|\alpha_{j-1}(C)|$ of vertices adjacent to C having degree $j-1$ with respect to C. At each step the selection is passed by those vertices which do not increase too much the number $|\alpha_{j-1}(C)|$ of vertices adjacent to C having degree $j-1$ with respect to C, where "too much" means no more than t times the average value over all candidates at step j, for some suitable choice of t.

We will prove that this strategy causes the number $|\alpha_h(C)|$ of vertices adjacent to C having degree h with respect to C to increase very slowly, thus ensuring that this number is less than 1 until at least $\frac{n_A}{\beta \Delta_B}$ vertices have been chosen, for a fixed constant β.

The algorithm is presented in Figure 1; from now on, C_i denotes the h-cluster at the end of step i, and $S_{i,j}$ the set of vertices, to be added to C_{i-1}, that passed the selection step j. Furthermore, the notation $\alpha_0(C_i)$ is extended to denote the set $B - \alpha(C_i)$ of all vertices in B not adjacent to C_i.

Lemma 2. $|S_{i,j}| \geq (n_A - i + 1)\left(1 - \frac{1}{t}\right)^j$.

Proof. At each step j we select those vertices $x \in S_{i,j-1}$ such that $|\alpha_{j-1}(C_{i-1}) \cap \alpha(x)|$ is no more than t times the average value $\mu_{i,j-1}$ over all vertices in $S_{i,j-1}$. If a set of n non-negative integers with average value μ then at most n/t elements have value greater than $t\mu$ and, hence, at least $n(1 - 1/t)$ elements have value at most $t\mu$, thus $|S_{i,j}| \geq (1 - 1/t)|S_{i,j-1}|$, with $|S_{i,0}| = n_A - i + 1$; the Lemma follows.

Lemma 3. *Let* $n_{i,j} \doteq |\alpha_j(C_i)|$, *that is the number of vertices* y *in* $\alpha(C_i)$ *s.t.* $\delta_{C_i}(y) = j$. *Then*

$$n_{i,j} \leq \left[\frac{t\Delta_B(i-1)}{(n_A - i + 1)\left(1 - \frac{1}{t}\right)^{\frac{j-1}{2}}} \right]^j \frac{n_B}{j!}$$

$C_0 \leftarrow \emptyset$;
$i \leftarrow 0$;
repeat
  $i \leftarrow i + 1$;
  $S_{i,0} \leftarrow A - C_{i-1}$;
  for $j \leftarrow 1$ **to** h **do begin**
    $\mu_{i,j-1} = \frac{\sum_{x \in S_{i,j-1}} |\alpha_{j-1}(C_{i-1}) \cap \alpha(x)|}{|S_{i,j-1}|}$;
    $S_{i,j} \leftarrow \{x \in S_{i,j-1} : |\alpha_{j-1}(C_{i-1}) \cap \alpha(x)| \leq t\mu_{i,j-1}\}$;
  end ;
  pick a vertex $x \in S_{i,j}$;
  $C_i \leftarrow C_{i-1} \cup \{x\}$;
until $S_{i,j} = \emptyset$;
end .

Fig. 1. Algorithm Select.

Proof. The proof is by induction on the step j.

Base step: $j = 1$. At step $(i, 1)$, $\mu_{i,0}$ is the average degree of the $n_A - i + 1$ vertices in $A - C_{i-1}$ with respect to vertices not connected to $\alpha(C_{i-1})$. Thus, $\mu_{i,0} \leq \frac{m}{n_A - i + 1}$, and a vertex x that is added to $S_{i,1}$ verifies $\delta_{\alpha_0(C_{i-1})}(x) \leq \frac{tm}{n_A - i + 1}$.

If x is added to C_{i-1}, $n_{i-1,1}$ is increased by at most $\frac{tm}{n_A - i + 1}$ new vertices. Hence,

$$n_{i,1} \leq n_{i-1,1} + \frac{tm}{n_A - i + 1} \leq \sum_{k=1}^{i-1} \frac{tm}{n_A - k} \leq \frac{tm(i-1)}{n_A - i + 1} \quad .$$

Since $m \leq \Delta_B n_B$, $n_{i,1} \leq \frac{t\Delta_B n_B (i-1)}{n_A - i + 1}$, and the base step is proved.

Induction step: $j - 1 \rightarrow j$. At step (i, j), $\mu_{i,j-1}$ is the average degree of candidate vertices in $S_{i,j-1}$ with respect to vertices in $\alpha_{j-1}(C_{i-1})$. By Lemma 2 and since the total number of edges outgoing from $\alpha_{j-1}(C_{i-1})$ is at most $\Delta_B n_{i-1,j-1}$, we have $\mu_{i,j-1} \leq \frac{\Delta_B n_{i-1,j-1}}{(n_A - i + 1)\left(1 - \frac{1}{t}\right)^{j-1}}$.

A vertex x is added to $S_{i,j-1}$ if it verifies $\delta_{\alpha_{j-1}(C_{i-1})}(x) \leq t\mu_{i,j-1}$. Hence, if x is added to C_{i-1}, $n_{i-1,j}$ is increased by at most $t\mu_{i,j-1}$ new vertices. Thus,

$$\begin{aligned}
n_{i,j} &\leq n_{i-1,j} + \frac{t\Delta_B n_{i-1,j-1}}{(n_A - i + 1)\left(1 - \frac{1}{t}\right)^{j-1}} \leq \sum_{k=1}^{i-1} \frac{t\Delta_B n_{k,j-1}}{(n_A - k)\left(1 - \frac{1}{t}\right)^{j-1}} \\
&\leq \frac{t\Delta_B}{(n_A - i + 1)\left(1 - \frac{1}{t}\right)^{j-1}} \sum_{k=1}^{i-1} n_{k,j-1} \\
&\leq \frac{t\Delta_B}{(n_A - i + 1)\left(1 - \frac{1}{t}\right)^{j-1}} \sum_{k=1}^{i-1} \left[\frac{t\Delta_B(k-1)}{(n_A - k + 1)\left(1 - \frac{1}{t}\right)^{\frac{j-2}{2}}}\right]^{j-1} \frac{n_B}{(j-1)!} \\
&\leq \left[\frac{t\Delta_B}{(n - i + 1)\left(1 - \frac{1}{t}\right)^{\frac{j-1}{2}}}\right]^{j} \frac{n_B}{(j-1)!} \sum_{k=1}^{i-1} (k-1)^{j-1} \\
&\leq \left[\frac{t\Delta_B}{(n - i + 1)\left(1 - \frac{1}{t}\right)^{\frac{j-1}{2}}}\right]^{j} \frac{n_B}{j!}(i-1)^j \ .
\end{aligned}$$

This concludes the induction step.

Theorem 4. *Let $G = (A \cup B, E)$ be a bipartite graph. For $h \geq \lfloor \log n_B \rfloor$, Algorithm Select finds a h-cluster C of $\left\lceil \frac{n_A}{(2e^{\frac{3}{2}}+1)\Delta_B} \right\rceil$ vertices in time $O(|C| n_A \Delta_A)$.*

Proof. If $t = h \geq 2$, then $\left(1 - \frac{1}{t}\right)^{\frac{h-1}{2}} \geq \frac{1}{\sqrt{e}}$. Let $i_{\max}$ be the value of index i at the end of the execution of Algorithm Select. Considering that $h! \geq \left(\frac{h}{e}\right)^h$, Lemma 3 implies:

$$n_{i_{\max},h+1} \leq \left[\frac{e^{\frac{3}{2}}\Delta_B(i_{\max}-1)}{n_A - i_{\max} + 1}\right]^{h+1} n_B \ .$$

If $i_{\max} < \left\lceil \frac{n_A}{2e^{\frac{3}{2}}\Delta_B + 1} \right\rceil$ then $\frac{e^{\frac{3}{2}}\Delta_B(i_{\max}-1)}{n_A - i_{\max}+1} < \frac{1}{2}$, hence, $n_{i_{\max},h+1} \leq \frac{n_B}{2^{h+1}} < 1$ for $h \geq \lfloor \log n_B \rfloor \geq \lceil \log n_B \rceil - 1$, and $C_{i_{\max}}$ is a h-cluster.

From now on β denotes the constant $2e^{\frac{3}{2}} + 1 < 10$. The following theorem will be used to derive the space complexity of the proposed data structure.

Theorem 4 leads to the following

Corollary 5. *Let $G = (A \cup B, E)$ be a bipartite graph. For $h \geq \lfloor \log n_B \rfloor$, A can be partitioned into $\lceil 2\beta\Delta_B \rceil \cdot \lceil \log n_A \rceil$ h-clusters. The time complexity is $n_A^2 \Delta_A$.*

Proof. The sequence of clusters is computed by repeatedly selecting a h-cluster and removing its vertices from A. Let us suppose that after k iterations the number n'_A of vertices remained in A is greater than $n_A/2$, but after $k+1$ iterations is less than or equal

$n_A/2$. By Theorems 4, during the first k iterations, algorithm Select finds h-clusters of at least $\left\lceil \frac{n_A/2}{\beta \Delta_B} \right\rceil$ vertices. Hence, in k iterations at least $k \left\lceil \frac{n_A/2}{\beta \Delta_B} \right\rceil$ vertices have been removed from A, so $k \leq 2\beta\Delta_B$.

We can repeat the same argument to the remaining vertices, each time halving the number of vertices still in A; this can obviously repeated no more than $\lceil \log n_A \rceil$ times.

3.3 Partitioning into Clusters

The following lemma characterizes the complexity of partitioning a bipartite graph $G = (A \cup B, E)$ into clusters (1-clusters). Clusters will be used to build the ground data structure upon which the others are based.

Lemma 6. *Let $G = (A \cup B, E)$ be a bipartite graph. B can be partitioned in $1 + \Delta_B(\Delta_A - 1)$ clusters. The time required is $O(n_B \Delta_A \Delta_B)$*

Proof. Let $B_1, \ldots, B_k$ be a partition of B into clusters so that B_i is a maximal cluster for $B - \bigcup_{j=1}^{i-1} B_j$. Each vertex $y \in B_i$ has at most Δ_B adjacent vertices, and each of them has at most $\Delta_A - 1$ adjacent vertices different from y. Hence, a vertex $y \in B_i$ prevents at most $\Delta_B(\Delta_A - 1)$ vertices to be included in the same cluster. Since the cluster is maximal, each vertex in B either has been chosen in B_i or has been excluded from it, so $n_B \leq |B_i|(1 + \Delta_B(\Delta_A - 1))$. Hence $|B_i| \geq \frac{n_B}{1+\Delta_B(\Delta_A-1)}$. The lemma follows.

Note that the bound given by Lemma 6 is tight, since there exists an infinite class of regular bipartite graphs that cannot be decomposed in less than $1+\Delta(\Delta-1)$ clusters [11].

4 The Data Structure

In this Section we present the data structure for the multi–dimensional searching problem.

Based upon the decomposition theorems given in Section 3.2, we previously present a data structure for labeled bipartite that allows us to represent a bipartite graph $G = (A \cup B, E)$ in $O(n + m \log n)$ space, and to test if two vertices are adjacent with a constant number of steps. For sake of clarity, we first describe a simpler data structure that represent bi-regular bipartite graphs, then extend the result to represent all bipartite graphs.

4.1 Representing Bi-Regular Bipartite Graphs

Given a bi-regular bipartite graph $G = (A \cup B, E)$, we partition A in h-clusters according to Corollary 5; hence, we obtain a sequence of at most $\lceil 2\beta\Delta_B h \rceil$ bipartite graphs $G_i = (A_i, B, E_i)$, where A_i is the i-th h-cluster and $E_i \doteq E \cap (A_i \times B)$. Then we partition the vertex set B of each bipartite $G_i = (A_i \cup B, E_i)$ into clusters. Lemma 6 ensures that each bipartite graph is decomposed into at most $1 + h(\Delta_A - 1)$ clusters.

We define the following arrays:

- `hclus` of size n_A; $i =$ `hclus`$[x]$ is the index of the unique h-cluster A_i to which $x \in A$ belongs;

- `clus` of size $n_B \times \lceil 2\beta\Delta_B h \rceil$; $j = \texttt{clus}[y, i]$ is the index of the unique cluster $B_{i,j}$ in G_i to which $y \in B$ belongs;
- $\texttt{join}_i$ of size $n_A \times (1 + h(\Delta_A - 1))$; $y = \texttt{join}[x, j]$ is the unique possible vertex $y \in B$ adjacent to x in the j-th cluster in the unique i-th h-cluster to which x belongs.

Adjacency on the bipartite graph can be tested in 3 steps since $(x, y) \in E$ if and only if, given $i \doteq \texttt{hclus}[x]$ and $j \doteq \texttt{clus}[y, i]$, $y = \texttt{join}[x, j]$ holds. The total space required is

$$O(n_A + n_B \lceil 2\beta\Delta_B h \rceil + n_A(1 + h(\Delta_A - 1))) = O((n + m) \log n) \ .$$

Note that if $m \leq n$ then isolated vertices can be trivially represented, so the space complexity becomes $O(n + m \log n)$.

4.2 Representing Bipartite Graphs and 2–place Functions

We now show how to obtain for general bipartite graphs the same results as for bi-regular graphs. Given a bipartite graph $G = (A \cup B, E)$, we first partition B into maximal subsets B_i, s.t. $\forall y \in B_i$, $2^i \leq \delta(y) < 2^{i+1}$. We obtain a sequence of at most $h = \lceil \log n \rceil$ bipartite graphs $G_i = (A \cup B_i, E_i)$, where B_i is the i-th subset of B and $E_i \doteq E \cap (A \times B_i)$.

Then, according to Corollary 5, for each such bipartite graph G_i we partition A into h-clusters $A_{i,j}$, obtaining a sequence of at most $\left\lceil 2^{i+1}\beta \right\rceil h$ bipartite graphs $G_{i,j}$, and further partition each h-cluster into at most h subsets $A_{i,j,k}$ s.t. $\forall x \in A_{i,j,k}$, $2^k \leq \delta_{B_i}(x) < 2^{k+1}$, obtaining a sequence of bipartite graphs $G_{i,j,k}$.

Finally, for each bipartite graph $G_{i,j,k}$, we partition the set B_i into clusters; Lemma 6 ensures that each bipartite graph $G_{i,j,k}$ is decomposed into at most $1 + h(\Delta_{A_{i,j,k}} - 1)$ clusters.

We define the following arrays:

- `range` of size n_B; $i = \texttt{range}[y]$ is the index of the unique subset B_i to which y belongs;
- `hclus` of size $n_A \times h$; $j = \texttt{hclus}[x, i]$ is the index of the unique h-cluster $A_{i,j}$ to which $x \in A$ belongs in G_i.
- `subs` of size $n_A \times h$; $k = \texttt{subs}[x, i]$ is the index of the unique subset $A_{i,j,k}$ in the unique h-cluster to which $x \in A$ belongs in G_i.
- For each vertex $y \in B_i$, we define an array $\texttt{ranges}_y$ of size $\left\lceil 2^{i+1}\beta \right\rceil h$; $\texttt{ranges}_y[j]$ is a reference to the array `clus`, which contains the cluster indices of y in all subsets $A_{i,j,k}$; it is empty if y is not adjacent to any vertex in $A_{i,j}$. The total space needed for array $\texttt{ranges}_y[j]$ for all $y \in B$ is

$$O(\sum_{B_i} \sum_{y \in B_i} \left\lceil 2^{i+1}\beta \right\rceil h) = O(\sum_{B_i} \sum_{y \in B_i} \delta(y)\beta h) = O(mh) \ .$$

  Reading $\texttt{ranges}_y[j]$ requires 2 steps, one to read the initial address of the array given y, and one to access its j-th element.
- For each vertex $y \in B_i$, and each h-cluster $A_{i,j}$ connected to y, we define an array `clus`; $\texttt{clus}[k]$ is the index of the unique cluster in $G_{i,j,k}$ to which $y \in B_i$ belongs; it is empty if y is not adjacent to any vertex in $A_{i,j,k}$. For each vertex $y \in B_i$, since

$2^i \leq \delta(y) < 2^{i+1}$, at most 2^{i+1} such arrays are defined, each of them having size h. Hence, the total space needed for all arrays `clus` is

$$O(\sum_{B_i} \sum_{y \in B_i} 2^i h) = O(\sum_{B_i} \sum_{y \in B_i} \delta(y) h) = O(mh) \ .$$

- `joins` of size $n_A \times h$; `joins`$[x, i]$ is a reference to the array `join`, which contains all vertices in B_i adjacent to x. It is empty if x is not adjacent to any vertex in B_i.
- For each vertex $x \in A_{i,j,k}$, and each set B_i connected to x, we define an array `join` of size $(1 + h(2^{k+1} - 1))$; `join`$[l]$ is the (unique) possible vertex adjacent to x in the l-th cluster of $G_{i,j,k}$; it is empty if x is not adjacent to any vertex the l-th cluster of $G_{i,j,k}$. For each vertex $x \in A$, the space needed for all its related arrays `join` is $O(\sum_{B_i} 2\delta_{B_i}(x)) = O(h\delta(x)$, so the total space for arrays `join` for all $x \in A$ is $O(mh)$.

Adjacency on the bipartite graph can be tested in constant time since $(x, y) \in E$ if and only if, given $i \doteq$ `range`$[y]$, $j \doteq$ `hclus`$[x, i]$, $k \doteq$ `subs`$[x, i]$, `clus` $\doteq$ `ranges`$_y[j]$, $l \doteq$ `clus`$[k]$ and `join` $\doteq$ `joins`$[x, i]$, $y =$ `join`$[l]$ holds. The test requires 7 steps. The total space required is

$$O(n_B + n_A h + mh) = O((n + m) \log n) \ .$$

Also in this case, if $m \leq n$ then isolated vertices can be trivially represented, so the space complexity becomes $O(n + m \log n)$.

¿From the above discussion, we have the following theorem:

Theorem 7. *There exists a data structure that represents a bipartite graph with n vertices and m edges in space $O(n + m \log n)$. Vertex adjacency can be tested in 7 steps. Preprocessing time is $O(n^2 \Delta)$, where Δ is the maximum vertex degree.*

The representation of a 2–place function and the lookup operation which given two objects, return a value associated to the pair, is equivalent to the following: given a bipartite graph $G = (A \cup B, E)$, and a *labeling function* $\mathcal{L} : E \to \mathbb{N}$, and $x \in A$, $y \in B$, if $(x, y) \in E$ return $\mathcal{L}(x, y)$. This can be easily accomplished with the previously described data structure and, whenever $(x, y) \in E$, extending `join`$[l]$ to contain both the (unique) possible vertex y adjacent to x in the l-th cluster of $G_{i,j,k}$ and the value $\mathcal{L}(x, y)$. This leads to the following theorem:

Theorem 8. *There exists a data structure that represents a 2–place function of size m between objects from a domain of size n in space $O(n + m \log n)$. The lookup operation requires 7 steps. Preprocessing time is $O(n^3)$.*

4.3 Representing Multi–dimensional Data

Given a point set $S \subseteq U^d$, with $m = |S|$ and $n = |U|$. Let $\langle f_1, \ldots, f_{d-1} \rangle$ be the sequence of 2–place functions representing S, as described in Section 2. Additionally, let $m = |S|$ and m_i be the size of the 2–place function f_i, for $1 \leq i \leq d - 1$. By the definition, we have $m = \sum_{i=1}^{d-1} m_i$. Moreover, for any i, $n_i \leq m_i$, n_i being the size of the domain set of f_i. By Theorem 8, we can state the following theorem:

Theorem 9. *There exists an implicit data structure that represents a set $S \subseteq U^d$ of multi–dimensional points in space $O(m \log n)$. The exact match and prefix–partial match queries can be performed in $7(d-1)$ and $7(d-1)+t$ steps, respectively, where t is the number of points reported. The preprocessing time is $O(dn^3)$.*

5 Extensions

5.1 External Memory Data Structure

Due to its nature, the above described data structure can be efficiently applied to secondary storage. In this paper, we consider the standard two-level I/O model introduced by Aggarwal and Vitter in [1]. In this case, we can devise a powerful compression technique leading to a space optimal data structure.

In the data structure described in Section 4.2, the critical arrays are `ranges`, `clus`, and `join`, that is those requiring a total space $O(mh)$, which in terms of external memory storage implies $O(mh/B)$ blocks. The following lemma counts the number of non-empty entries in these arrays:

Lemma 10. *Let k' be the total number of non-empty entries in arrays $\texttt{ranges}_y$ for all $y \in B$; k'' be the total number of non-empty entries in all arrays `clus`; and k''' be the total number of non-empty entries in arrays `join`. Then $k' \leq m$, $k'' \leq m$, $k''' \leq m$.*

Proof. If $\texttt{ranges}_y[j]$ is not empty, then some edge (x, y) belongs to $G_{i,j}$; on the other hand, there is a unique bipartite graph $G_{i,j}$ containing such edge. Hence, the total number of non-empty entries in arrays $\texttt{ranges}_y$ for all $y \in B$ is at most m.

If $\texttt{clus}[k]$ is not empty for some vertex $y \in B_i$ and some h-cluster $A_{i,j}$ connected to y, then some edge (x, y) belongs to $G_{i,j,k}$; since there is a unique bipartite graph $G_{i,j,k}$ containing such edge, the total number of non-empty entries in all arrays `clus` is at most m.

If $\texttt{join}[l]$ is not empty, then some edge (x, y) belongs to $G_{i,j,k}$; there is a unique bipartite graph $G_{i,j,k}$ containing such edge; the total number of non-empty entries in arrays `join` for all $x \in A$ is at most m.

Let `a` be an array of size k. We partition `a` into intervals of B elements, and represent each interval by a reference to the block containing the non empty entries in that interval. It is easy to see that an array `a` of size k with k' empty entries can be represented in $O(\frac{k}{B} + k')$ space, thus in $O(\frac{k}{B^2} + \frac{k'}{B})$ blocks; furthermore, one access to $\texttt{a}[i]$ maps to 2 memory accesses. hence the external memory version of Theorem 8 and Theorem 9 can be stated as follows:

Theorem 11. *There exists an external–memory data structure that represents a 2–place function of size m between objects from a domain of size n with $O(\frac{n \log n+m}{B})$ blocks. The lookup operation requires 10 I/Os.*

Theorem 12. *There exists an external memory implicit data structure that represents a set $S \subseteq \mathbb{N}^d$ of multi–dimensional points with $O(m/B)$ blocks. The exact match and prefix–partial match queries require $10(d-1)$ and $10(d-1)+t/B$ I/Os, respectively, where t is the number of points reported by the partial match query.*

5.2 Incremental Exact Match Queries

The representation we propose for a multi–dimensional point set S allows to efficiently perform the exact match operation in a more general context. In fact we can define the *incremental exact match* query, where the coordinates are specified incrementally, that is, the search starts when the first coordinate is given, and proceeds refining the searching space as soon as the other coordinates are specified. This definition of exact search is particularly useful in distributed environments where the request for a query is expressed by sending messages along communication links [15,16,4,12,13] and not all coordinates reside on the same machine.

Another field of application of the incremental exact match query is for the *interactive exploratory search* on Web. In this case the user can specify the searching keys one by one so as to obtain intermediate results.

Also, the incremental exact match query is particularly practical when dealing with a point sets from a very high multi–dimensional space (order of thousands of keys). In this case we can manage the query in a distributed environment by specifying only $k \ll d$ keys a time in order to prevent network congestion and to obtain a more reliable answer.

5.3 Improving Conventional Searching Data Structures

Our decomposition technique can be positively applied to one-dimensional hashing and perfect hashing when dealing with real keys. Let w be the machine word length, and $K \gg w$ the key length. We can divide each key in K/w sub-keys, and reduce the original one–dimensional searching problem to a multi–dimensional searching problem, which can be solved with our technique with no additional storage and with a constant number of I/Os.

Another important application it to the trie data structure. With a technique similar to the one above described, we can consider larger node sizes.

6 Open Problems

One important open problem is that of dynamizing the data structure; even an incremental-only version data structure would be a useful improvement. Another important research direction is to extend the operation set to include other operations useful for the management of a multi-dimensional data set (e.g. range queries, retrieve maximal elements, orthogonal convex-hull, *etc*.)

We are currently carrying out an extensive experimentation on secondary memory, based on a data sets derived from a business application. This experimentation activity is still at its beginning, the main purpose being primarily to test the effective speedup in the lookup operation and the overall size of the representation on these data sets. Preliminary experimentation results show that the behavior of our data structure is very fast and works very well in the average case.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.
2. R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationship in large data and knowledge bases. In *Proceedings of the International Conference on the Management of Data*, pages 253–262, Portland, OR, 1989.
3. R. Bayer and C. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–179, 1972.
4. R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *4th Int. Conf. on Foundations of Data Organization and Algorithms (FODO)*, Chicago, 1993.
5. D.Mayer and B.Vance. A call to order. In *Proceedings of the International Conference on Principle of Database Systems*, 1993.
6. A. Fiat and M. Naor. Implicit $O(1)$ probe search. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 336–344, Seattle, Washington, 1989.
7. A. Fiat, M. Naor, J. P. Schmidt, and A. Siegel. Non-oblivious hashing. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing: Chicago, Illinois, May 2–4, 1988*, pages 367–376, New York, NY 10036, USA, 1988. ACM Press.
8. M. L. Fredman, J. Komlós, and E. Szemeredi. Sorting a sparse table with $O(1)$ worst case access time. In *Proc. 23rd Ann. IEEE Symp. on Foundations of Computer Science*, pages 165–169, 1982.
9. T. Hagerup. Sorting and searching on the word RAM. In M. Morvan, C. Meinel, and D. Krob, editors, *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398. LNCS 1373, Springer-Verlag, 1998.
10. P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Science*, 52:589–612, 1996.
11. D. J. Kleitman and K. J. Winston. The asymptotic number of lattices. *Annuals of Discrete Matemathics*, 6:243–249, 1980.
12. B. Kröll and P. Widmayer. Distributing a search tree among a growing number of processors. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 265–276, Minneapolis,MN, 1994.
13. B. Kröll and P. Widmayer. Balanced distributed search trees do not exists. In S. Akl et al., editor, *4th Int. Workshop on Algorithms and Data Structures (WADS'95)*, pages 50–61, Kingston, Canada, 1995. LNCS 955, Springer-Verlag.
14. Douglas Lea. Digital and Hilbert K-D trees. *Information Processing Letters*, 27(1):35–41, 1988.
15. W. Litwin, M. A. Neimat, and D. A. Schneider. LH^*–linear hashing for distributed files. In *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 1993.
16. W. Litwin, M. A. Neimat, and D. A. Schneider. LH^*–a scalable distributed data structure. *ACM Trans. Database Systems*, 21(4):480–525, 1996.
17. D. Morrison and R. Patricia. Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.
18. F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, Berlin, New York, 1985.
19. S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database System*, pages 25–35, 1994.
20. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

21. M. Talamo and P.Vocca. Compact implicit representation of graphs. In J. Hromkovič and O. Sýkora, editors, *Proceedings of* 24*th International Workshop on Graph–Theoretic Concepts in Computer Science WG'98*, pages 164–176. LNCS 1517, Springer-Verlag, 1998.
22. M. Talamo and P. Vocca. A data structure for lattice representation. *Theoretical Computer Science*, 175(2):373–392, April 1997.
23. M. Talamo and P. Vocca. A time optimal digraph browsing on a sparse representation. Technical Report 8, Matemathics Department, University of Rome "Tor Vergata", 1997.
24. A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, Redwood City, CA, 1993.
25. M. Yannakakis. Graph-theoretic methods in database theory. In ACM, editor, *PODS '90. Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems: April 2–4, 1990, Nashville, Tennessee*, volume 51(1), New York, NY 10036, USA, 1990. ACM Press.