# Chapter 3.5 Hash Structure of Multi-Dimensional Data

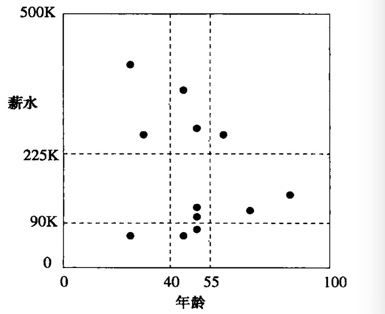
Build hash function using single key gets another two data structures:

* Grid File:Sort values of the dimension to divide this dimension.
* Piecewise Hash Function: Hash each dimensions and every dimension does influence the bucket.

## Chapter 3.5.1 Grid File

The simplest data structure compared with single index (hash function with single key) is Grind File. (The space be divided as the grind line space).

***Principles:***

* Grind line separates the dimension into strip, the dots on the grind line will be considered belongs to the strip which is lower than this grind line.
* Different grind can have different numbers of grind line. Interval length can be different within the neighboring grind lines.
* The same grind line can have different interval length.

***Example:***

Insert customer with two properties (age, salary) into two-dimension space:

|  |  |  |  |
| --- | --- | --- | --- |
| (25, 60) | (45, 60) | (50, 75) | (50, 100) |
| (50, 120) | (70, 110) | (85, 140) | (30, 260) |

Look from the right image, twelve points are located in a single two-dimensional space. In every single dimensional, two lines are used to divide the current dimension into three parts. In sum, there are nine spaces in the current two-dimension space.

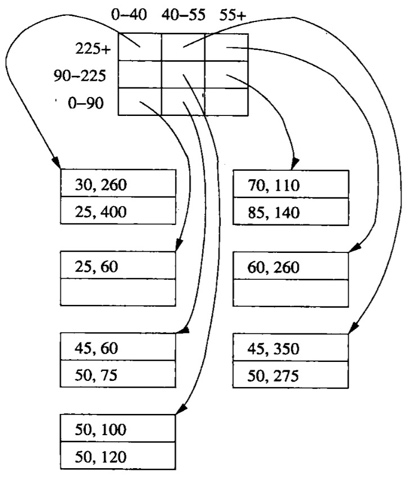
* 225K <= salary <= 500K, 0 <= age < 40;
* 225K <= salary <= 500K, 40 <= age < 55;
* 225K <= salary <= 500K, 55 <= age <= 100;
* 90K <= salary < 225K, 0 <= age < 40;
* 90K <= salary < 225K, 40 <= age < 55;
* 90K <= salary < 225K, 55 <= age <= 100;
* 0K <= salary < 90K, 0 <= age < 40;
* 0K <= salary < 90K, 40 <= age < 55;
* 0K <= salary < 90K, 55 <= age < 100;

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 – 40 | 40 – 55 | 55 + |
| 0k – 90k | (25, 60k) | (45, 60k)  (50, 75k) | Empty |
| 90k – 225k | Empty | (50, 100k)  (50, 120k) | (70, 110k)  (85, 140k) |
| 225k – 500k | (30, 260k)  (25, 400k) | (45, 350k)  (50, 275k) | (60, 260k) |

## Chapter 3.5.2 Searching on Grid File

***Principles:***

* See every space as a bucket of hash table, and every single record in each space is seen as the record of this bucket.
* In order to locate a record, the value of every dimension should be confirmed. We focus on every record and its location in the grid space.
* The location of every single dimension decides its location in its bucket.

***Example:***

The bucket here is a 3\*3 matrix which is used to indicate nine spaces.

* Two empty spaces.
* Two records at most in every single bucket.
* Overflow blocks are un-necessary if no bucket has exceeded two records.

## Chapter 3.5.3 Insertion in Grid File

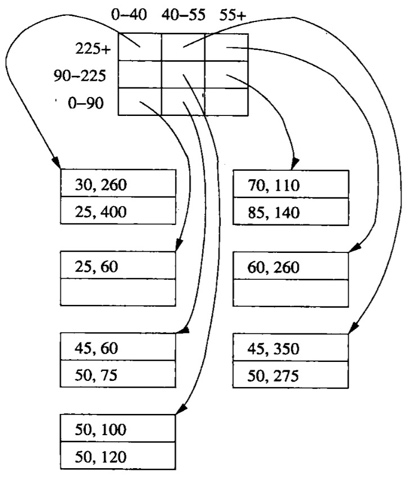
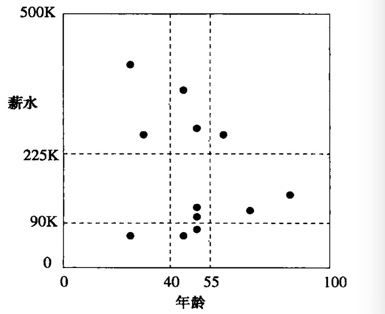
***Description:***

When record is needed to be inserted into a grid file, then we search the bucket and put the record into the bucket.

***Principle:***

* Find space in the bucket. -> Nothing has to be done.
* Can not find space in the bucket.
  + Add overflow blocks to the bucket.
  + Add or remove grid line to reconstruct structure.

Construct an index for the single dimension if the number of strip is very large. The searching key of index should be a set which is used to divide the current dimension.

***Example:***

Customer A: (52, 200K)

The record needs to be inserted into the middle of space. Read from the space, we can tell that there already have two records. The first one is (50, 100) and the second one is (50, 120).

The insertion of new customer (52, 200K) will exceed the limited value of bucket.

***Two solutions:***

1. Add an overflow block for the bucket. The bucket will include three records, which contains:

(50, 100K), (50, 120K), (52, 200K)

1. Choose a grid line to spilt the bucket, either in age or salary dimension. Two records in one side while another one in another.

* Vertical line, such as age = 51. It separates the customer with the age 50 and 52.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 – 40 | 40 – 51 | 51 – 55 | 55 + |
| 0k – 90k | (25, 60k) | (45, 60k)  (50, 75k) | Empty | Empty |
| 90k – 225k | Empty | (50, 100k)  (50, 120k) | (52, 200k) | (70, 110k)  (85, 140k) |
| 225k – 500k | (30, 260k)  (25, 400k) | (45, 350k)  (50, 275k) | Empty | (60, 260k) |

* Horizontal line, such as salary = 130k. It separates the customer with the salary 120k and 200k.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 – 40 | 40 – 55 | 55 + |
| 0k – 90k | (25, 60k) | (45, 60k)  (50, 75k) | Empty |
| 90k – 130k | Empty | (50, 100k)  (50, 120k) | (70, 110k) |
| 130k – 225k | Empty | (52, 200k) | (85, 140k) |
| 225k – 500k | (30, 260k)  (25, 400k) | (45, 350k)  (50, 275k) | (60, 260k) |

* Horizontal line, such as salary = 115k. It separates the customer with the salary 100k and 120k, 110k and 140k.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 – 40 | 40 – 55 | 55 + |
| 0k – 90k | (25, 60k) | (45, 60k)  (50, 75k) | Empty |
| 90k – 115k | Empty | (50, 100k) | (70, 110k) |
| 115k – 225k | Empty | (50, 120k)  (52, 200k) | (85, 140k) |
| 225k – 500k | (30, 260k)  (25, 400k) | (45, 350k)  (50, 275k) | (60, 260k) |

## Chapter 3.5.4 Grid File Performance

***Description:***

Consider the performance of searching from grid file.

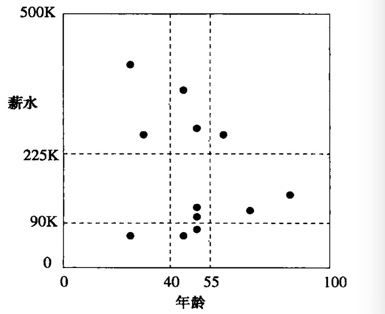
***Principle:***

If data distribution is great in grid file, and the data file is not large, then we need to choose the grid line to make:

1. Create less bucket and then store bucket into main memory. Then when searching from the bucket or add a new row/column into the matrix by inserting a new line into bucket by inserting a new line into two-dimension space will not cause disk I/O.
2. Store index of every single dimension grid line into the main memory or avoid using index but using binary-search to find the value in the main memory.
3. The bucket has only little overflow blocks, so when searching the buckets, it will not cause too much I/O.

***Specific Conditions:***

* Search specific record
* Find an appropriate bucket -> read operation. (one disk I/O)
* Insert or delete record -> write operation. (one disk I/O)
* Create an overflow block -> write operation. (one disk I/O)
* Search partial matching (Find customers with age = 50 Or Find customers with salary = 200k.)
* Find one row or column or row of all buckets. (Decided by bucket numbers.)
* Search range matching (Find customers with age range in (25 – 40) and with salary range in (50k – 100k).)
* Range searching includes a wide range of buckets and the number of bucket that needs to check will not exceed the result set too much.
* Search the nearest neighbor
* For point P, need to find the belonging bucket, considering there exists point Q. But if in the neighbor place, the distance between point P and the edge of a bucket is less than P to Q, then this bucket is needed to be searched.

*Example:*

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 – 40 | 40 – 55 | 55 + |
| 0k – 90k | (25, 60k) | (45, 60k)  (50, 75k) | Empty |
| 90k – 225k | Empty | (50, 100k)  (50, 120k)  *(45, 200k)* | (70, 110k)  (85, 140k) |
| 225k – 500k | (30, 260k)  (25, 400k) | (45, 350k)  (50, 275k) | (60, 260k) |

Find the nearest point to P (45, 200k), according to the graph and chart above:

* The point Q (50, 120k) is the nearest point whose distance is 80.2 in the current bucket.
* In the first line, the largest salary is 90k. 200 – 90 = 110 which is larger than 80.2. Therefore, skip check the first line.
* In the bucket range (90k – 225k) and (55 +), (225k – 500k) and (0 – 40), (40 – 55) and (225k – 500k), (55 +) and (225k – 500k), (0 – 40) and (90k – 225k) all needs to be checked.
* Two points (30, 260k) and (60, 260k) are the same distance from point P, which is 61.8.

## Chapter 3.5.5 Piecewise Hash Function

***Principle:***

* Hash function accepts an attribute list as function parameter.
* Design hash function, which generates several binary bits, such as . Generate hash value for property which makes .
* Hash function is a list of () , which means hash function applies on the property, then generates binary bit.
* Property list (), need to calculate , and get the final sequence list.

***Example1:***

For 1024 buckets which means 10 bits’ number buckets, assign 4 bits to property a and other 6 bits to property b.

Assume that property(a) = A and property(b) = B. Also = 0101 and = 111000.

So the bucket these two properties should placed is 0101 111000, just concatenate of two binary sequences.

***Example2:***

*Description:*

Assume data list below, we save data into a hash function segmented list which contains 8 buckets and means there has three bits to represent these 8 buckets. Assume each block can store two records. Then give one bit to age property and the other two give to salary property.

(Combine one bit decided by age property and two bits decided by salary property to determine the final location in the segment list.)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 – 40 | 40 – 55 | 55 + |
| 0k – 90k | (25, 60k) | (45, 60k)  (50, 75k) | Empty |
| 90k – 225k | Empty | (50, 100k)  (50, 120k) | (70, 110k)  (85, 140k) |
| 225k – 500k | (30, 260k)  (25, 400k) | (45, 350k)  (50, 275k) | (60, 260k) |

*Solution:*

applied on age, = mod 2. The remainder (0, 1) is used to decide the location of record.

* Remainder: 0 -> 0XY
* Remainder: 1 -> 1XY

applied on salary, = mod 4. The remainder (0, 1, 2, 3) is used to decide the location of record.

* Remainder: 0 -> X00
* Remainder: 1 -> X01
* Remainder: 2 -> X10
* Remainder: 3 -> X11

*Calculation:*

(25, 60k): 25 mod 2 = 12...1 and 60 mod 4 = 0...0 -> location of (25, 60k) is 100

(45, 60k): 45 mod 2 = 22...1 and 60 mod 4 = 0...0 -> location of (45, 60k) is 100

(50, 75k): 50 mod 2 = 25...0 and 75 mod 4 = 18...3 -> location of (50, 75k) is 011

(50, 100k): 50 mod 2 = 25...0 and 100 mod 4 = 25...0 -> location of (50, 100k) is 000

(50, 120k): 50 mod 2 = 25...0 and 120 mod 4 = 30...0 -> location of (50, 120k) is 000

(70, 110k): 70 mod 2 = 35...0 and 110 mod 4 = 27...2 -> location of (70, 110k) is 010

(85, 140k): 85 mod 2 = 42...1 and 140 mod 4 = 35...0 -> location of (85, 140k) is 100

(30, 260k): 30 mod 2 = 15...0 and 260 mod 4 = 65...0 -> location of (30, 260k) is 000

(25, 400k): 25 mod 2 = 12...1 and 400 mod 4 = 100...0 -> location of (25, 400k) is 100

(45, 350k): 45 mod 2 = 22...1 and 350 mod 4 = 87...2 -> location of (45, 350k) is 110

(50, 275k): 50 mod 2 = 25...0 and 275 mod 4 = 68...3 -> location of (50, 275k) is 011

(60, 260k): 60 mod 2 = 30...0 and 260 mod 4 = 65...0 -> location of (60, 260k) is 000

|  |  |  |
| --- | --- | --- |
| Bucket No | Bucket | Extended Bucket |
| 000 | (50, 100k)  (50, 120k) | (60, 260k)  (30, 260k) |
| 001 |  |  |
| 010 | (70, 110k) |  |
| 011 | (50, 75k)  (50, 275k) |  |
| 100 | (25, 60k)  (25, 400k) | (45, 60k)  (85, 140k) |
| 101 |  |  |
| 110 | (45, 350k) |  |
| 111 |  |  |

## Chapter 3.5.6 Compare Between Grid File and Piecewise Hash Function

Performance difference between Grid File and Piecewise Hash Function

* The Piecewise Hash Function does not make any sense to the nearest neighbor search and range search, the problem is that the distance between two points can not be presented by physical distance.
* Choose a good hash function to hash points to each buckets, and these buckets will be seen occupied evenly. For grid file, especially the dimensions that are big enough, it will be easier to leave some empty buckets or almost empty bucket. The Piecewise Hash Function will leave less buckets or generate less overflow blocks.