Multi-Dim. Index Structures:

- Hash-table like
  - Grid Files
  - Partitional Hashing

- Tree-like
  - Multipl
  - Key Indexes
  - k-d tree
  - Quad tree
  - R tree

This set of notes...
Grid Index (Grid Files)

A Grid Index (File) is organized into a 2-dimensional structure:

Key

\[
\begin{array}{cccc}
  V_1 & V_2 & V_3 & \cdots & V_m \\
  X_1 & X_2 & X_3 & \cdots & X_n \\
\end{array}
\]

This is a file and stored e.g. as follows:

```
\[ m/n \quad V_1 \quad V_2 \quad \cdots \quad V_m \quad X_1 \quad X_2 \quad \cdots \quad X_n \]
```

Pointer to records with (key1 = V_3, key2 = X_2).
In general, the keys are ranges or "buckets".

E.g.:

- [235k, +)
- [90k, 235k)
- [0, 90k)
- [0, 40), [40, 55), [55, +)

* Ranges are called "Grid Lines"
Grid Index File:

- Uses an array whose dimension is the same as for the data file.
- Store - for each dimension - the list of values at which the "grid lines" occur.
- Uses overflow buckets: blocks.
Example index:

- Suppose we have a bunch of records of people buying jewelry.
- Assume the only relevant attributes are:
  
  \[(\text{name, address, age, salary})\]

  Important properties to categorize buyers.

- We want to index the information on:
  
  \[(\text{age, salary})\].

Data (input):

\[
\begin{align*}
A & : (25, 6) & D & : (45, 60) & G & : (50, 75) & J & : (50, 100) \\
B & : (50, 120) & E & : (70, 110) & H & : (85, 140) & K & : (30, 260) \\
C & : (25, 400) & F & : (45, 350) & I & : (50, 175) & L & : (60, 260)
\end{align*}
\]

Ranges:

- Age: \(0 - 40, 40 - 55, 55 - \#
- Salary: \(0 - 90K, 90K - 225K, 225K - \#

Grid Index File for these rows:

Data Blocks

A(25, 60)
D(45, 60)
G(50, 75)

The book represents as follows:

- Grid Index File
- Data Blocks
- A(25, 60)
- D(45, 60)
- G(50, 75)
Lookup in a Grid Index File

Recall: Grid Index file contains:

1. Number of dimensions

2. List of values at which the grid lines occur in each dimension.

Given: a "point" = value in N-dimension $\langle v_1, v_2, v_3 \ldots, v_N \rangle$.

=> determine the "position" of the "point" in the grid for each dimension.
Example:

<table>
<thead>
<tr>
<th>Age</th>
<th>0-40</th>
<th>40-55</th>
<th>55+</th>
</tr>
</thead>
<tbody>
<tr>
<td>225+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90-125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0-90</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Given point:

- age = 25 \rightarrow 0-40
- salary = 60 \rightarrow 0-90
Inserting into a Grid Index File

(1) Do lookup first
   \rightarrow find blue points.

(2) If there is room in data block
   \Rightarrow insert

Overflow:

(A) Add overflow block

(B) Reorganize the grid lines.

It's like "dynamic hashing"
but more complex.

\checkmark

New grid line will divide
MANY buckets!'

(Could make many small/empty buckets!)}
Performance of Grid Index

Assumptions:

1. The bucket matrix \( \text{grid of bln points} \)

   - \( \text{grid in memory} \)

2. Values of the grid lines

   can also be kept in memory.

3. Few overflow blocks.

   (50 access to data block = 1 blk)
Performance of Grid Index - Numbers.

1. Look up a specific point $(v_1, v_2, ..., v_N)$
   - $N$ blk access to get blk pointer
   - $1$ blk access to get data block.

   **Insert/Delete:**
   May require an additional Write blk.
   (or overflow).

2. Partial Match query: $(v_1, *)$

   - $N$ blk access to get blk pointers

   ![Diagram](attachment:image.png)
Partial Match search: (example)

\[ \text{age} = 50 \quad \text{Salary} = * \]

\[ \text{swen} \quad \text{unknown} \]

Records for age = 50 are found here:

- Look up specific point: very fast
  (it's hashing after all...)
3. Range Queries:

All data blocks in the range.

4. Nearest Neighbor Query:

About: 9 data blocks

Caveat
It actually depends on the geometry.

If the grid is very stretched:

\[ \text{Nearest Neighbor} \]
Partitioned Hashing:

Ordinary Hashing:

\[ V \rightarrow h(\cdot) \rightarrow h(v) \]

- Key
- Hash function
- Hash value (used to find record with key = v)

Partitioned Hashing:

Key is composite: \((v_1, v_2, \ldots, v_n)\)

Uses \(n\) hash functions:

\[ h_1(\cdot), h_2(\cdot), \ldots, h_n(\cdot) \]

Hash value = concatenation of

\[ h_1(v_1) \cdot h_2(v_2) \cdot \ldots \cdot h_n(v_n). \]
Example:

\[ v_1 = a \quad \downarrow \quad h(a) \quad \downarrow \quad 0101 \]

\[ v_2 = b \quad \downarrow \quad h(b) \quad \downarrow \quad 111000 \]

\[ \text{Hash value.} \]

10 bits (1024 buckets)

- Difference between Ordinary Hashing vs Partitioned Hashing:

  1. Ordinary Hashing cannot hash when one (or more) keys are missing.

  Partitioned Hashing:

  \[ v_1 = a \quad v_2 = ? \quad \alpha \text{ unknown} \]

  \[ \downarrow \]

  \[ h(a) \]

  \[ 0101 \]

  \[ \ldots \ldots \]

  \( \Rightarrow \) you can still get a RANGE (subset) of buckets to search!
Concrete Example: who buys jewelry data

\[ h_1(x) = \text{age} \mod 2 \]
\[ h_2(x) = \left( \frac{\text{salary}}{1000} \right) \mod 4. \]

3 bit hash value

Assume: 2 records per block

Result:

- 30,260
- 50,120
- 60,260
- 70,110
- 75,60
- 45,60
- 85,140
- 45,350
- Range queries:

- Hash the values in the range \((v_1, v_2) \in R\) → \(h(v_1, v_2)\)
- Use the \(h(v_1, v_2)\) to access the blocks.

- Nearest Neighbor:

  Partitioned Hashing is

  **completely useless** for Nearest Neighbor

  (because there is no notion of distance in the hash function.

  closeness of the bucket numbers ≠ physical distance between data "points" (records).
Advantages of Partitioned Hashing:

- Good hash function can randomize the records:
  - Good occupancy rate (avg # records per bucket similar for all buckets).

- Good files (especially with LARGE # dimensions) will have many empty buckets.
Handling Tiny Buckets:

Instead:

You can pack different buckets into one block:

Block header must contain information on:

1. bucket ID
2. records in specific bucket.

Eg:

[Diagram of a block structure with records and a header containing bucket IDs and record counts.]