Hashing

- basic idea: compute from the key K directly the location of
  - the record or
  - the bucket (containing pointers to several records)
- hash function $h(K) \Rightarrow$ bucket number
- then: store a pointer to the record with key K in the bucket $h(K)$

- hash function $h(key)$ returns address of bucket or record
- initially: buckets are stored successively
- if the keys for a specific hash value do not fit into one block then use a chain of overflow blocks
  - additional I/Os to find the desired pointer
  - leave space initially for later insertions

How Do We Choose a Hashing Function?

- Desired property: expected number of keys per bucket is the same for all buckets (uniform distribution)
  - and is relatively small
  - ideally all buckets consist of only one page
- Example of a bad hash function for a string attribute
  - first three letters of the string value
- A potentially good hash function
  - $(\text{char}_1 + \text{char}_2 + \ldots + \text{char}_N) \mod b$, where $b$ is the number of buckets
- See Knuth Vol. 3 for more on good hashing functions
How Many Buckets?

• Keep space utilization between 50% and 80%:
  – utilization = #keys used / total #keys that fit
• How do we cope with growth? (the #keys may be difficult to predict)
  ⇒ double the number of buckets, the redistribute entries (problem: read/write the whole file)
  ⇒ use **extensible hashing**:
    – use another level of indirection for buckets ("directory")
    – double just the directory and split only the bucket that overflowed
    – some bits of h(K) are used to address into the directory

Extensible Hashing

• use i of f bits output by hash function/ as grows over time as an address in the bucket directory

Extensible Hashing Example

locate entry with h(K) = 5 (=101 binary)
Extensible Hashing

- Insertions: if space just insert, else
  - split bucket
  - if local-depth < global-depth then local-depth++
  - else double directory (global-depth++)

- Collisions (data entries with the same value) may required overflow pages!
Indexing vs Hashing

- Hashing is more efficient for probes given key
  - SELECT ... FROM R WHERE R.A=5
- Indexing (conventional and B+trees) handles range queries
  - SELECT ... FROM R WHERE R.A>5 AND R.A<10
- So far: only one-dimensional indexes

Why Multidimensional Indexes?

- Spatial data (point and region data):
  - Geographic Information Systems (GIS) containing points, lines, regions, ...
  - Types of spatial queries:
    - partial match queries (all records where X=55)
    - range queries
      - 50<salary<60 AND 40<age<50 (*)&
      - all rivers in Wisconsin
    - nearest neighbor (10 ATMs closest to X,Y)
    - spatial join (all cities near a lake)
- Q: How about KEY = (Salary,Age) for (*)?

Why Multidimensional Indexes?

- Data cubes: for decision support, OLAP (online analytic processing) systems can store aggregations in multiple dimensions
  - find sales of pink shirts for each store and month in 1999
- Multimedia databases: e.g., find similar images (based on feature vectors => nearest neighbor in the feature space)
  => cluster (pointers to) “similar” objects (i.e., close to each other in the multidimensional space)
Multidimensional Queries in SQL

• Nearest neighbor(s) of (10,20) in SQL

```
SELECT * FROM POINTS p
WHERE NOT EXISTS
    (SELECT * FROM POINTS q
     WHERE (p.x-10)^2 + (p.y-10)^2 + (q.x-20)^2 + (q.y-20)^2 <
         (p.x-20)^2 + (p.y-10)^2 + (p.x-10)^2 + (p.y-20)^2);
```

• Find sales of pink shirts for each store and month

```
SELECT store, COUNT(*) FROM sales
WHERE item="shirt" AND color="pink"
GROUP BY store, month
```

Multi-Key Indexing

• Motivation: queries of the form
  - SELECT ... FROM R
  WHERE cond1 AND cond2
  - cond1 and cond2 are equality or range conditions

• Solution 1: use index for only one of the conditions
  - suggested if there is a very selective condition

• Solution 2: pointer intersection
  - fairly selective conditions

```
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year > 3
```

Rewriting & Optimization

```
π Name
σ Dept="Toys" AND Year>3 Employee
```

Find employees of the Toys dept with >3 years in the company

```
SELECT Name FROM Employee
WHERE Dept="Toys" AND Year > 3
```

<table>
<thead>
<tr>
<th>Dept Index</th>
<th>Year Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toys</td>
<td>1</td>
</tr>
<tr>
<td>PCs</td>
<td>2</td>
</tr>
<tr>
<td>Pens</td>
<td>3</td>
</tr>
<tr>
<td>Suits</td>
<td>4</td>
</tr>
<tr>
<td>Dept Index</td>
<td>5</td>
</tr>
<tr>
<td>Year Index</td>
<td>1</td>
</tr>
<tr>
<td>Toys</td>
<td>2</td>
</tr>
<tr>
<td>PCs</td>
<td>3</td>
</tr>
<tr>
<td>Pens</td>
<td>4</td>
</tr>
<tr>
<td>Suits</td>
<td>5</td>
</tr>
<tr>
<td>Year Index</td>
<td>2</td>
</tr>
<tr>
<td>Toys</td>
<td>3</td>
</tr>
<tr>
<td>PCs</td>
<td>4</td>
</tr>
<tr>
<td>Pens</td>
<td>5</td>
</tr>
<tr>
<td>Suits</td>
<td>1</td>
</tr>
<tr>
<td>Year Index</td>
<td>3</td>
</tr>
</tbody>
</table>

Pointer Intersection (Solution 2)
A Spatial Range Query Using B-Trees

- answer = \{(X,Y,Z) \mid db(X,Y,Z), 450<X,Y<550\}
  - |db|=1,000,000, 100 records/disk block
  - |within X|=within Y|= 100,000, |answer|=10,000
- B-tree indexes for X and for Y (200 pointers/leaf)
- #Disk I/Os:
  - access 100,000 B-tree leaves in X and in Y dim.
    \Rightarrow 2\times 500 \text{ disk I/Os}
  - retrieve 10,000 answer records
    \Rightarrow likely to access almost all disk blocks!

Q: What if selectivity =1% (instead of 10%)?

Solution 3: Multi-Key Indexing with Grid Files

Grid File

- Space overhead (very sparse structure)
- Insertion and deletion can be expensive
Spatial Queries with Grid Files

- **Lookup:**
  - find records where age=45, salary=90:
    - using scales in memory identify grid cell and get it (from disk)
    - grid entry points to disk block containing the desired records
- **Partial match:**
  - similar to lookup (not all dimensions may be given)
- **Range queries:**
  - find records with $45 < \text{age} < 50$ and $90 \leq \text{salary} < 100$:
    - using scales, find all grid cells, search only those blocks
- **Nearest neighbor queries:**
  - search the pointed to block/bucket and possible those adjacent ones which may have a closer point
  - may even involve searching non-adjacent blocks (Why?)

Inserting into Grid Files

- 3 records/block

Inserting into Grid Files

- insert 4 => split along X and redistribute

Inserting into Grid Files

- insert 5 => split A along Y => A, C (also splits the region pointing to B)
- insert 6 => ok

Inserting into Grid Files

- insert 7 => ok
- insert 8 => split C along X => C, D
- insert 9 => ok
### Other approaches

- **Partitioned hash functions:**
  - $h(A,B) \rightarrow h_1(A) \cdot h_2(B)$ (concat bits of hashed attributes)
  - (+) partial match queries: find where $A = a$ (using $h_1$ we know where to look)
  - (-) nearest neighbor, range queries

- **Other approaches** (supporting range, nearest neighbor)
  - **Multiple-key indexes:**
    - use different attributes at different levels
    - **Quad trees** (recursively subdivide into quadrants)
  - **R-trees** (region trees: hierarchy of regions, based on bounding boxes, regions may overlap)