Dynamic Hashing

- Periodic rehashing
  - If number of entries in a hash table becomes (say) 1.5 times size of hash table,
    - create new hash table of size (say) 2 times the size of the previous hash table
    - Rehash all entries to new table

- Linear Hashing
  - Do rehashing in an incremental manner

- Extendable Hashing
  - Tailored to disk based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets
Extendable Hashing

- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically $b$-bit integers, with $b = 32$.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be $i$ bits, $0 \leq i \leq 32$.
    - Bucket address table size = $2^i$. Initially $i = 0$
    - Value of $i$ grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket (why?)
  - Thus, actual number of buckets is $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure

In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)
Use of Extendable Hash Structure

- Each bucket $j$ stores a value $i_j$
  - All the entries that point to the same bucket have the same values on the first $i_j$ bits.

- To locate the bucket containing search-key $K_j$:
  1. Compute $h(K_j) = X$
  2. Use the first $i_j$ high order bits of $X$ as a displacement into bucket address table, and follow the pointer to appropriate bucket

- To insert a record with search-key value $K_j$
  - follow same procedure as look-up and locate the bucket, say $j$.
  - If there is room in the bucket $j$ insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)
    - Overflow buckets used instead in some cases (will see shortly)

Insertion in Extendable Hash Structure (Cont.)

To split a bucket $j$ when inserting record with search-key value $K_j$:

- If $i > i_j$ (more than one pointer to bucket $j$)
  - allocate a new bucket $z$, and set $i_z = i_j + 1$
  - Update the second half of the bucket address table entries originally pointing to $j$, to point to $z$
  - remove each record in bucket $j$ and reinsert (in $j$ or $z$)
  - recompute new bucket for $K_j$ and insert record in the bucket (further splitting is required if the bucket is still full)

- If $i = i_j$ (only one pointer to bucket $j$)
  - If $i$ reaches some limit $b$, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment $i$ and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for $K_j$
      Now $i > i_j$ so use the first case above.
Deletion in Extendable Hash Structure

- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of $i_j$ and same $i_j - 1$ prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Example (Cont.)

- Initial hash structure; bucket size = 2
Example (Cont.)

- Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records

```
<table>
<thead>
<tr>
<th>hash prefix</th>
<th>bucket address table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1515: Mozart   Music  40000</td>
</tr>
<tr>
<td></td>
<td>10121: Srinivasan Comp. Sci. 90000</td>
</tr>
<tr>
<td></td>
<td>12121: Wu      Finance 90000</td>
</tr>
</tbody>
</table>
```

Example (Cont.)

- Hash structure after insertion of Einstein record

```
<table>
<thead>
<tr>
<th>hash prefix</th>
<th>bucket address table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1515: Mozart   Music  40000</td>
</tr>
<tr>
<td></td>
<td>12121: Wu      Finance 90000</td>
</tr>
<tr>
<td></td>
<td>22222: Einstein Physics 95000</td>
</tr>
<tr>
<td></td>
<td>10101: Srinivasan Comp. Sci. 65000</td>
</tr>
</tbody>
</table>
```
Example (Cont.)

- Hash structure after insertion of Gold and El Said records

Example (Cont.)

- Hash structure after insertion of Katz record
Example (Cont.)

And after insertion of eleven records

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Department</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>58583</td>
<td>Califiere</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
</tbody>
</table>

Example (Cont.)

And after insertion of Kim record in previous hash structure

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Department</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>Finance</td>
<td>80000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
</tr>
<tr>
<td>58583</td>
<td>Califiere</td>
<td>History</td>
<td>62000</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
</tr>
</tbody>
</table>
Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead

- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B*-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation

- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Bloom Filters

- A bloom filter is a probabilistic data structure used to check membership of a value in a set
  - May return true (with low probability) even if an element is not present
  - But never returns false if an element is present
  - Used to filter out irrelevant sets

- Key data structure is a single bitmap
  - For a set with \( n \) elements, typical bitmap size is \( 10n \)

- Uses multiple independent hash functions

- With a single hash function \( h() \) with range=number of bits in bitmap:
  - For each element \( s \) in set \( S \) compute \( h(s) \) and set bit \( h(s) \)
  - To query an element \( v \) compute \( h(v) \), and check if bit \( h(v) \) is set

- Problem with single hash function: significant chance of false positive due to hash collision
  - 10% chance with \( 10n \) bits
Bloom Filters (Cont.)

- Key idea of Bloom filter: reduce false positives by use multiple hash functions $h_i()$ for $i = 1..k$
  - For each element $s$ in set $S$ for each $i$ compute $h_i(s)$ and set bit $h_i(s)$
  - To query an element $v$ for each $i$ compute $h_i(v)$, and check if bit $h_i(v)$ is set
    - If bit $h_i(v)$ is set for every $i$ then report $v$ as present in set
    - Else report $v$ as absent
  - With $10n$ bits, and $k = 7$, false positive rate reduces to 1% instead of 10% with $k = 1$

Write Optimized Indices

- Performance of B+-trees can be poor for write-intensive workloads
  - One I/O per leaf, assuming all internal nodes are in memory
  - With magnetic disks, < 100 inserts per second per disk
  - With flash memory, one page overwrite per insert
- Two approaches to reducing cost of writes
  - Log-structured merge tree
  - Buffer tree
Log Structured Merge (LSM) Tree

- Consider only inserts/queries for now
- Records inserted first into in-memory tree ($L_0$ tree)
- When in-memory tree is full, records moved to disk ($L_1$ tree)
  - B*-tree constructed using bottom-up build by merging existing $L_1$ tree with records from $L_0$ tree
- When $L_1$ tree exceeds some threshold, merge into $L_2$ tree
  - And so on for more levels
  - Size threshold for $L_{i+1}$ tree is $k$ times size threshold for $L_i$ tree

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number $n$ it must be easy to retrieve record $n$
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits
Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute:
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

- Example

```
<table>
<thead>
<tr>
<th>record number</th>
<th>ID</th>
<th>gender</th>
<th>income_level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>76766</td>
<td>m</td>
<td>L1</td>
</tr>
<tr>
<td>1</td>
<td>22222</td>
<td>f</td>
<td>L2</td>
</tr>
<tr>
<td>2</td>
<td>12121</td>
<td>f</td>
<td>L1</td>
</tr>
<tr>
<td>3</td>
<td>15151</td>
<td>m</td>
<td>L4</td>
</tr>
<tr>
<td>4</td>
<td>58583</td>
<td>f</td>
<td>L3</td>
</tr>
</tbody>
</table>
```

- Bitmap indices are useful for queries on multiple attributes:
  - Not particularly useful for single attribute queries

- Queries are answered using bitmap operations:
  - Intersection (and)
  - Union (or)

- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap:
  - E.g., $100110 \text{ AND } 110011 = 100010$
    - $100110 \text{ OR } 110011 = 110111$
    - $\text{NOT } 100110 = 011001$
  - Males with income level L1: $10010 \text{ AND } 10100 = 10000$
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster
Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g., if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size

Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
  - E.g., 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B⁺-trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
  - Above technique merges benefits of bitmap and B⁺-tree indices
Spatial Data

- Databases can store data types such as lines, polygons, in addition to raster images
  - allows relational databases to store and retrieve spatial information
  - Queries can use spatial conditions (e.g. contains or overlaps).
  - queries can mix spatial and nonspatial conditions
- Nearest neighbor queries, given a point or an object, find the nearest object that satisfies given conditions.
- Range queries deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- Spatial join of two spatial relations with the location playing the role of join attribute.

R-Trees

- R-trees are a N-dimensional extension of B⁺-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R⁺-trees and R* -trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B⁺-tree node to an N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case \((N = 2)\)
  - generalization for \(N > 2\) is straightforward, although R-trees work well only for relatively small \(N\)
- The bounding box of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - Bounding boxes of children of a node are allowed to overlap
Example R-Tree

- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.

Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched, but works acceptably in practice.