Indexing and Hashing
Why Indices?

- Table Students (id, name, major, address)
- Query 1: find student with id 200830782
  - Scanning the table to find the student – $O(n)$ time, can be costly
  - Sort all students on id into a sorted list, conduct a binary search on the sorted list – $O(\log n)$ query time, fast
    - What if the sorted list cannot be held into main memory?
- Query 2: find students majoring in computer science
  - Can the sorted list on id help?
  - Sort all students on major into a sorted list, conduct a binary search on the sorted list – $O(n \log n)$ construction time, $O(\log n)$ query time, $O(n)$ space
- Some issues
  - An update to table Students has to be propagated to both sorted lists
  - Tradeoff between time and space – we cannot afford to construct a separate sorted list for each query
  - Queries may be raised ad hoc – we may not gain to construct a separate sorted list on the fly for each query
What Are Indices?

- An index is an (efficient) data structure that can facilitate answering a set of queries
  - General – can be used to answer a set of queries
  - Efficient – construction, query answering, space, and maintenance
- Issues in index construction
  - Query types – what kinds of queries that an index can support
  - Query answering time
  - Construction time and space cost
  - Maintenance cost
- Search key – an attribute or a set of attributes used to look up records in a file
  - An index is built to facilitate searching on a search key
  - A search key may not be unique – different from key in database design
Indices Can Make Big Difference

SELECT *
FROM Table1 Table2
WHERE P1 AND P2
– P1 and P2 are on Table1 and Table2, respectively
• Table1 and Table2 contain 1 million tuples each, P1(Table1) and P2(Table2) contain 100 tuples each
• Without index, $10^{12}$ tuples will be read!
• With index, only 10,000 tuples will be read!
Primary and Secondary Indices

- Primary indices (clustering indices) – records are sequentially ordered in the search key order
- Secondary indices (nonclustering indices) – search keys specify an order different from the sequential order of the records
- Index-sequential files

<table>
<thead>
<tr>
<th>Brighton</th>
<th>A-217</th>
<th>Brighton</th>
<th>750</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mianus</td>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>Redwood</td>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td></td>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>A-305</td>
<td>Round Hill</td>
<td>350</td>
</tr>
</tbody>
</table>
Dense and Sparse Indices

- **Dense index:** an index record appears for every search-key value in the file
- **Sparse index:** an index record appears for only some of the search-key values
- **Search algorithms**
  - Tradeoff between space and query answering efficiency

<table>
<thead>
<tr>
<th>Dense Index</th>
<th>Sparse Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>A-217 Brighton 750</td>
</tr>
<tr>
<td>Downtown</td>
<td>A-101 Downtown 500</td>
</tr>
<tr>
<td>Mianus</td>
<td>A-110 Downtown 600</td>
</tr>
<tr>
<td>Perryridge</td>
<td>A-215 Mianus 700</td>
</tr>
<tr>
<td>Redwood</td>
<td>A-102 Perryridge 400</td>
</tr>
<tr>
<td>Round Hill</td>
<td>A-201 Perryridge 900</td>
</tr>
<tr>
<td></td>
<td>A-218 Perryridge 700</td>
</tr>
<tr>
<td></td>
<td>A-222 Redwood 700</td>
</tr>
<tr>
<td></td>
<td>A-305 Round Hill 350</td>
</tr>
</tbody>
</table>
Two-level Sparse Indices

- When the index is too big, we can divide it into multiple levels
- Structure
- Search algorithm
Index Maintenance

- Insertion
  - Dense indices
  - Sparse indices
  - Multilevel indices
- Deletion
  - Dense indices
  - Sparse indices
  - Multilevel indices
- Reserving space for evolution
Secondary Indices

• Each table can have only one primary index
• Secondary indices are needed if multiple search keys are needed in a table
• A dense index where the records pointed to by successive values in the index are not sorted sequentially
Search and Maintenance

• Search algorithm
  – Clustering read to reduce I/O cost – sort disk addresses of tuples and read in sequence

• Maintenance
  – Insertion
  – Deletion

• Multilevel secondary index

• When a new tuple is inserted or deleted into a table, all indices on the table need to be updated
B⁺-Tree

- The most widely used index structure
- Idea: maintain a balanced tree which can serve as a sorted list
- Binary search tree: an inorder tree walk results in a sorted list
- Balanced versus unbalanced tree
  - Searching a balanced search tree $O(\log n)$
  - Searching an unbalanced search tree can be $O(n)$
Why and When Balanced Trees?

- If a balanced tree can be held into main memory, is it a good choice?
- If a balanced tree cannot be held into main memory, is it a good choice?
- Each node is a disk block
  - An I/O access retrieves the whole block into main memory
- Within a block, data are organized as a sorted list
### Leaf Nodes of a B⁺-Tree

- For $i=1, 2, \ldots, n-1$, pointer $P_i$ points to:
  - A file record with search key value $K_i$, or
  - A bucket of pointers, each of which points to a file record with search key value $K_i$
- Each leaf node holds at least $\lceil (n - 1) / 2 \rceil$, up to $(n-1)$ values

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$K_1$</th>
<th>$P_2$</th>
<th>...</th>
<th>$P_{n-1}$</th>
<th>$K_{n-1}$</th>
<th>$P_n$</th>
</tr>
</thead>
</table>

![Leaf node diagram](image)

- Account file
- Leaf node:

<table>
<thead>
<tr>
<th>ID</th>
<th>Location</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-212</td>
<td>Brighton</td>
<td>750</td>
</tr>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Non-leaf Nodes

- A non-leaf node may hold up to \( n \) pointers, must hold at least \( \left\lfloor n/2 \right\rfloor \) pointers – at least half full
  - Fanout: the number of pointers in a node
- For \( i = 2, 3, \ldots, n-1 \), pointer \( P_i \) points to the subtree that contains search key values less than \( K_i \) and greater than or equal to \( K_{i-1} \)
- Pointer \( P_n \) points to the subtree that contains those key values greater than or equal to \( K_{n-1} \)
- Pointer \( P_1 \) points to the part of the subtree that contains those search key values less than \( K_1 \)
Root Node

- Can hold fewer than \( \lceil n/2 \rceil \) pointers
- If a tree consists of more than one node, the root node must hold at least two pointers

\[
\begin{array}{c}
\text{N=3} \\
\text{Brighton} \quad \text{Downtown} \quad \text{Mianus} \quad \text{Perryridge} \quad \text{Redwood} \quad \text{Round Hill}
\end{array}
\]

\[
\begin{array}{c}
\text{N=5} \\
\text{Brighton} \quad \text{Downtown} \quad \text{Mianus} \quad \text{Perryridge} \quad \text{Redwood} \quad \text{Round Hill}
\end{array}
\]
Query Answering on B+-tree

- Given value V, find tuples whose search key has value V
- Set current-node = root node
- While current-node is not a leaf node do
  - Let K_i be the smallest search key value, if any, greater than V
  - If there is no such value then let n be the number of pointers in the node, set current-node to the node pointed to by P_n
  - Else set current-node to the node pointed to by P_i
- If there is a key value K_i in current-node such that K_i = V, then pointer P_i points to the desired record or bucket; else no record with key value V exists
- Query answering efficiency
  - If there are K search key values in the file, the tree is no taller than $\lfloor \log_{\frac{n}{2}} K \rfloor$
  - Let n = 100, K = 1 million, each query will search up to only 4 nodes!
  - For balanced binary tree, it may require about 20 block access on average per query – Why can a B+-tree save?
Insertion on B⁺-trees

- **Case 1:** the leaf node has enough space to hold a new pointer
  - Find the leaf node in which the search key value would appear
  - If the search key value already appears in the leaf node
    - Add the new record to the file
    - Add to the bucket a pointer to the record
  - If the search key value does not appear, but the leaf node still has space to hold one more pointer
    - Insert the value in the leaf node, adjust the search keys and pointers in the leaf node so that the search keys are still in order
  - Insert the record into the file, create the appropriate pointer
- **Case 2:** a leaf node needs to be split if it already has \((n-1)\) pointers
  - Split the leaf node into two
    - The first one (original one) has \(\lceil n/2 \rceil\) pointers
    - The second one (new one) has the remaining \(n - \lceil n/2 \rceil \geq \lceil (n-1)/2 \rceil\) pointers
  - A new pointer is inserted into the parent node
    - If the parent node has no space to hold the new pointer, split it
    - The recursive splitting ends at the root node
Splitting a Leaf Node: Example

Original B+-tree

Insertion of “Clearview”

Split of leaf node
Deleting a Leaf Node

Deleting “Downtown”
Coalescing Sibling Nodes

Deleting “Perryridge”

Diagram showing the process of deleting a node named Perryridge from a tree structure.
A Complicated Example

Deleting “Perryridge”
B-Tree Index Files

- Similar to B$^+$-tree, but B-tree allows search-key values to appear only once
  - Eliminates redundant storage of search keys
- Search keys in non-leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a non-leaf node must be included
  - Non-leaf node – pointers $B_i$ are the bucket or file record pointers

Leaf node

| $P_1$ | $K_1$ | $P_2$ | ... | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |

Non-leaf node

| $P_1$ | $B_1$ | $K_1$ | $P_2$ | $B_2$ | $K_2$ | ... | $P_{m-1}$ | $B_{m-1}$ | $K_{m-1}$ | $P_m$ |
B-tree Example

B-tree

B+-tree
B-Trees versus B+-Trees?

• Advantages of B-trees
  – May use less tree nodes than a corresponding B+-Tree
  – Sometimes possible to find search-key value before reaching leaf node

• Disadvantages of B-trees
  – Only a small fraction of all search-key values are found early
  – Non-leaf nodes are larger, so fan-out is reduced. B-Trees typically have greater depth than the corresponding B+-Trees
  – Insertion and deletion more complicated
  – Implementation is harder

• Typically, advantages of B-Trees do not outweigh disadvantages
B⁺-tree File Organization

- Even though the records having the same search key value are indexed by pointers, they still can be scattered in many blocks.
- B⁺-tree file organization: organize the blocks containing the actual records using the leaf nodes in the B⁺-tree.
- Index-organized tables in Oracle 10g.

![Diagram of a B⁺-tree with data nodes labeled with letters and values, and pointers connecting the nodes.]
Using Order Indices

SELECT loan_number FROM account
WHERE branch_name = “Perryridge” AND balance = 1000

• If there is an index on branch_name, then find all records satisfying
  branch_name = “Perryridge”, for each such a record, check whether
  balance = 1000
  – Similarly, we can also use an index on balance
• If there are two indices on branch_name and balance, respectively,
  how can we use them simultaneously?
  – What if there are many tuples in branch Perryridge, and many tuples of
    balance 1000, but very few in branch Perryridge have balance 1000?
• Composite search key: build an ordered index (e.g., a B+-tree) on
  (branch_name, balance)
  SELECT loan_number FROM account WHERE branch_name = “Perryridge”
  SELECT loan_number FROM account
  WHERE branch_name < “Perryridge” AND balance = 1000
Miscellaneous Issues

• Variable-size buckets in leaf nodes complicates the B+-tree implementation
  – Adding an extra unique attribute (e.g., record-id) to the search key to make the search key unique

• In table Account (account_number, balance, branch_name, manager), if most of the time, we query balance according to account_number
  – In addition to an index on account_number, storing balance together with account_number speeds up the query answering
  – Covering indices are the ones that store the values of some selected attributes other than the search-key attributes along with the pointers to the record – tradeoff between query answering time and space (B+-tree file as the extreme)

• If a primary index is built on account_number, and a secondary index is built on balance, then an insertion/update of a tuple results in moves of tuples and thus the secondary index needs to be updated if pointers are used
  – Store account_number instead of pointers in the secondary index
  – Search in the secondary index
Hashing

- A hash table is an effective data structure for implementing dictionaries.
- In database systems, we can obtain the address of the disk block containing a desired record directly by computing a hash function on the search key value of the record.
- **Hash function**: a function from the set of all search key values to the set of all bucket addresses.
  - Bucket: a unit of storage that can store one or more records, typically a disk block, can be larger or smaller.

[Diagram: Search key → Disk address]
Operations Using Hashing

- Insertion: insert the tuple into bucket $h(K)$
  - What if the bucket overflows?
- Lookup: check every tuple in bucket $h(K)$
- Deletion: search bucket $h(K)$ for the target tuple, remove the tuple

Requirements
- The distribution is **uniform**
  - Assign each bucket the same number of search key values from the set of all possible search key values
- The distribution is **random**
  - In the average case, each bucket will have nearly the same number of tuples assigned to it, regardless of the actual distribution of search key values
  - Not correlated to any externally visible ordering on the search key values, such as alphabetic ordering or ordering by the length of the search keys
Bad/Good Hash Functions

• Map names by the first letters in the names
  – Simple, but not uniform
  – More names beginning with B and R than Q and X

• Map the balance in range 1 to 100k into 10 buckets
  \([1,10k), [10k, 20k), \ldots, [90k, 100k]\)
  – Uniform, since each bucket has the same number of different balance values
  – Not random, since records with balances between 1 to 10k are far more common than records with balances between 90k and 100k

• Suppose 10 buckets, the hash function returns the sum of the binary representations of the characters modulo 10
  – Uniform and random in expectation
Example

\[ h(\text{Perryridge}) = 5 \]
\[ h(\text{Round Hill}) = 3 \]
\[ h(\text{Brighton}) = 3 \]
Overflow Buckets

- Insufficient buckets
  - Assumption: number of buckets > total number of tuples / the number of tuples fit in a bucket
  - As the total number of tuples increases, the assumption may not hold

- Skew: some buckets are assigned more records than the others in a database instance
  - Multiple records may have the same search key
  - The chosen hash function may result in non-uniform distribution of search keys in a database instance

- Reduce the Probability of Overflow
  - Choose the number of buckets as 
    \[(1+d) \times \text{total number of tuples} / \text{number of tuples per bucket}\]
    - **Fudge factor** \(d\) is typically around 0.2
  - Trade off space against probability of overflow
Closed versus Open Hashing

- **Overflow chaining**: the overflow buckets of a given bucket are chained together in a linked list
  - Closed hashing

- **Open hashing** does not use overflow buckets
  - Use the available space in some other buckets
  - Generally not suitable for database applications
Hash Indices

- Organize the search keys and pointers into a hash file structure
Deficiencies of Static Hashing

- In **static hashing**, function $h$ maps search-key values to a fixed set $B$ of bucket addresses
  - Databases grow with time. If the initial number of buckets is too small, performance will degrade due to too much overflows
  - If file size at some point in the future is anticipated and the number of buckets allocated accordingly, a significant amount of space will be wasted initially
  - If database shrinks, again, space will be wasted
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive
- Dynamically modify the number of buckets?
Example

hash prefix

00 · · ·
01 · · ·
10 · · ·
11 · · ·
· · ·
· · ·

bucket address table

bucket 1

bucket 2

bucket 3
Example

<table>
<thead>
<tr>
<th>A-217</th>
<th>Brighton</th>
<th>750</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td>A-305</td>
<td>Round Hill</td>
<td>350</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>branch-name</th>
<th>h(branch-name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>0010 1101 1111 1011 0010 1100 0011 0000</td>
</tr>
<tr>
<td>Downtown</td>
<td>1010 0011 1010 0000 1100 0110 1001 1111</td>
</tr>
<tr>
<td>Mianus</td>
<td>1100 0111 1110 1101 1011 1111 0011 1010</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1111 0001 0010 0100 1001 0011 0110 1101</td>
</tr>
<tr>
<td>Redwood</td>
<td>0011 0101 1010 0110 1100 1001 1110 1011</td>
</tr>
<tr>
<td>Round Hill</td>
<td>1101 1000 0011 1111 1001 1100 0000 0001</td>
</tr>
</tbody>
</table>
Insert the First Record

<table>
<thead>
<tr>
<th>Branch Code</th>
<th>Branch Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-217</td>
<td>Brighton</td>
<td>750</td>
</tr>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td>A-305</td>
<td>Round Hill</td>
<td>350</td>
</tr>
</tbody>
</table>

```
branch-name | h(branch-name)
Brighton    | 0010 1101 1111 1011 0010 1100 0011 0000
Downtown    | 1010 0011 1010 0000 1100 0110 1001 1111
Mianus      | 1100 0111 1110 1101 1011 1111 0011 1010
Perryridge  | 1111 0001 0010 0100 1001 0011 0110 1101
Redwood     | 0011 0101 1010 0110 1100 1001 1110 1011
Round Hill  | 1101 1000 0011 1111 1001 1100 0000 0001
```
## Insert the Second/Third Records

<table>
<thead>
<tr>
<th>A-217</th>
<th>Brighton</th>
<th>750</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>branch-name</th>
<th>h(branch-name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>0010 1101 1111 1011 0010 1100 0011 0000</td>
</tr>
<tr>
<td>Downtown</td>
<td>1010 0011 1010 0000 1100 0110 1001 1111</td>
</tr>
<tr>
<td>Mianus</td>
<td>1100 0111 1110 1101 1011 1111 0011 1010</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1111 0001 0010 0100 1001 0011 0110 1101</td>
</tr>
<tr>
<td>Redwood</td>
<td>0011 0101 1010 0110 1100 1001 1110 1011</td>
</tr>
<tr>
<td>Round Hill</td>
<td>1101 1000 0011 1111 1001 1100 0000 0001</td>
</tr>
</tbody>
</table>

![Bucket Address Table and Hash Function Diagram]
## Insert the Fourth Record

<table>
<thead>
<tr>
<th>hash prefix</th>
<th>branch-name</th>
<th>h(branch-name)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Brighton</td>
<td>0010 1101 1111 1011 0010 1100 0011 0000</td>
</tr>
<tr>
<td></td>
<td>Downtown</td>
<td>1010 0011 1010 0000 1100 0110 1001 1111</td>
</tr>
<tr>
<td></td>
<td>Mianus</td>
<td>1100 0111 1110 1101 1011 1111 0011 1010</td>
</tr>
<tr>
<td></td>
<td>Perryridge</td>
<td>1111 0001 0010 0100 1001 0011 0110 1101</td>
</tr>
<tr>
<td></td>
<td>Redwood</td>
<td>0011 0101 1010 0110 1100 1001 1110 1011</td>
</tr>
<tr>
<td></td>
<td>Round Hill</td>
<td>1101 1000 0011 1111 1001 1100 0000 0001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Branch</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-217</td>
<td>Brighton</td>
<td>750</td>
</tr>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td>A-222</td>
<td>hash prefix</td>
<td></td>
</tr>
<tr>
<td>A-305</td>
<td>hash prefix</td>
<td></td>
</tr>
</tbody>
</table>

**Bucket Address Table**

- Bucket 1
  - A-217: Brighton 750
- Bucket 2
  - A-101: Downtown 500
  - A-110: Downtown 600
  - A-215: Mianus 700
Insert the Records of “Perryridge”

<table>
<thead>
<tr>
<th>branch-name</th>
<th>h(branch-name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brighton</td>
<td>0010 1101 1111 1011 0010 1100 0011 0000</td>
</tr>
<tr>
<td>Downtown</td>
<td>1010 0011 1010 0000 1100 0110 1001 1111</td>
</tr>
<tr>
<td>Mianus</td>
<td>1100 0111 1110 1101 1011 1111 0011 1010</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1111 0001 0010 0100 1001 0011 0110 1011</td>
</tr>
<tr>
<td>Redwood</td>
<td>0011 0101 1010 0110 1100 1001 1110 1011</td>
</tr>
<tr>
<td>Round Hill</td>
<td>1101 1000 0011 1111 1001 1100 0000 0001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record ID</th>
<th>Branch Name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-217</td>
<td>Brighton</td>
<td>750</td>
</tr>
<tr>
<td>A-101</td>
<td>Downtown</td>
<td>500</td>
</tr>
<tr>
<td>A-110</td>
<td>Downtown</td>
<td>600</td>
</tr>
<tr>
<td>A-215</td>
<td>Mianus</td>
<td>700</td>
</tr>
<tr>
<td>A-102</td>
<td>Perryridge</td>
<td>400</td>
</tr>
<tr>
<td>A-201</td>
<td>Perryridge</td>
<td>900</td>
</tr>
<tr>
<td>A-218</td>
<td>Perryridge</td>
<td>700</td>
</tr>
<tr>
<td>A-222</td>
<td>Redwood</td>
<td>700</td>
</tr>
<tr>
<td>A-305</td>
<td>Round Hill</td>
<td>350</td>
</tr>
</tbody>
</table>

bucket address table

hash prefix

1

2

3

A-217 Brighton 750
A-101 Downtown 500
A-110 Downtown 600
A-215 Mianus 700
A-218 Perryridge 700
A-102 Perryridge 400
A-201 Perryridge 900
Complete the Example

bucket address table

hash prefix

3

1

A-217 | Brighton | 750
A-222 | Redwood | 700

2

A-101 | Downtown | 500
A-110 | Downtown | 600

3

A-215 | Mianus | 700
A-305 | Round Hill | 350

3

A-102 | Perryridge | 400
A-201 | Perryridge | 900

A-218 | Perryridge | 700
Dynamic Hashing

- Allow the hash function to be modified dynamically
  - Good for databases that grow and shrink from time to time
- **Extensible hashing** – one form of dynamic hashing
  - The 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 values 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
  - Thus, actual number of buckets is $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets
Pros & Cons of Extensible Hashing

• Pros
  – Hash performance does not degrade with growth of file
  – Minimal space overhead

• Cons
  – Extra level of indirection to find desired record
  – Bucket address table may itself become very big (larger than memory)
    • Need a tree structure to locate desired record in the structure!
  – Changing size of bucket address table is an expensive operation
Ordered Indexing and Hashing

• Cost of periodic re-organization
• Relative frequency of insertions and deletions
• Is it desirable to optimize average access time at the expense of worst-case access time?
• Expected type of queries:
  – Hashing is generally better at retrieving records having a specified value of the key.
  – If range queries are common, ordered indices are preferred
Index Definition in SQL

- Not in SQL:1999 standard
- Create an index
  ```sql
  create index <index-name> on <relation-name> <attribute-list>)
  ```
  E.g.:
  ```sql
  create index b-index on branch(branch-name)
  ```
- Use `create unique index` to indirectly specify and enforce the condition that the search key is a candidate key
  - Not really required if SQL unique integrity constraint is supported
- To drop an index
  ```sql
  drop index <index-name>
  ```
### Bitmap Index

- For n tuples, a bitmap index has n bits and can be packed into \( \lceil n / 8 \rceil \) bytes and \( \lceil n / 32 \rceil \) words
- From a bit to the row-id: the j-th bit of the p-th byte \( \rightarrow \) row-id = p*8 +j

<table>
<thead>
<tr>
<th>cust</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack</td>
<td>M</td>
</tr>
<tr>
<td>Cathy</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Nancy</td>
<td>F</td>
</tr>
</tbody>
</table>

```
1 0 ... 0
```
Using Bitmap to Count

- Shcount[] contains the number of bits in the entry subscript
  - shcount[01100101]=4

```c
count = 0;
for (i = 0; i < SHNUM; i++)
  count += shcount[B[i]];
```
Advantages of Bitmap Index

- Efficient in space
- Ready for logic composition
  - $C = C_1 \text{ AND } C_2$
  - Bitmap operations can be used
- Bitmap index only works for categorical data with low cardinality
  - Naively, we need 50 bits per entry to represent the state of a customer in US
  - How to represent a sale in dollars?
Bit-Sliced Index

- A sale amount can be written as an integer number of pennies, and then represented as a binary number of N bits
  - 24 bits is good for up to $167,772.15, appropriate for many stores
- A bit-sliced index is N bitmaps
  - Tuple j sets in bitmap k if the k-th bit in its binary representation is on
  - The space costs of bit-sliced index is the same as storing the data directly
- Get the sum using a bit-sliced index
  - Get $c_i$, the count of each bit using the bitmap index
  - $\text{Sum}=\sum_i c_i \cdot 2^i$
Horizontal or Vertical Storage

- A fact table for data warehousing is often fat
  - Tens of even hundreds of dimensions/attributes
- A query is often about only a few attributes
- Horizontal storage: tuples are stored one by one
- Vertical storage: tuples are stored by attributes

<table>
<thead>
<tr>
<th>A_1</th>
<th>A_2</th>
<th>...</th>
<th>A_{100}</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_1</td>
<td>x_2</td>
<td>...</td>
<td>x_{100}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z_1</td>
<td>z_2</td>
<td>...</td>
<td>z_{100}</td>
</tr>
</tbody>
</table>
Horizontal Versus Vertical

- Find the information of tuple $t$
  - Typical in OLTP
  - Horizontal storage: get the whole tuple in one search
  - Vertical storage: search 100 lists

- Find $\text{SUM}(a_{100}) \text{ GROUP BY } \{a_{22}, a_{83}\}$
  - Typical in OLAP
  - Horizontal storage (no index): search all tuples $O(100n)$, where $n$ is the number of tuples
  - Vertical storage: search 3 lists $O(3n)$, 3% of the horizontal storage method

- Projection index: vertical storage