Chapter 11

Indexing & Hashing

1. Many queries reference only a small proportion of records in a file. For example, finding all records at Perryridge branch only returns records where *bname* = "Perryridge".
2. We should be able to locate these records directly, rather than having to read **every** record and check its branch-name. We then need extra file structuring.

11.1 Basic Concepts

1. An index for a file works like a catalogue in a library. Cards in alphabetic order tell us where to find books by a particular author.
2. In real-world databases, indices like this might be too large to be efficient. We'll look at more sophisticated indexing techniques.
3. There are two kinds of indices.
   - Ordered indices: indices are based on a sorted ordering of the values.
   - Hash indices: indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function, called a *hash function*.
4. We will consider several indexing techniques. No one technique is the best. Each technique is best suited for a particular database application.
5. Methods will be evaluated on:
   (a) **Access Types** — types of access that are supported efficiently, e.g., value-based search or range search.
   (b) **Access Time** — time to find a particular data item or set of items.
   (c) **Insertion Time** — time taken to insert a new data item (includes time to find the right place to insert).
   (d) **Deletion Time** — time to delete an item (includes time taken to find item, as well as to update the index structure).
   (e) **Space Overhead** — additional space occupied by an index structure.
6. We may have more than one index or hash function for a file. (The library may have card catalogues by author, subject or title.)
7. The attribute or set of attributes used to look up records in a file is called the **search key** (not to be confused with primary key, etc.).

Brighton	217	750
Downtown	101	500
Downtown	110	600
Mianus	215	700
Perriridge	102	400
Perriridge	201	900
Perriridge	218	700
Redwood	222	700
Round Hill	305	350

Figure 11.1: Sequential file for *deposit* records.

11.2 Ordered Indices

1. In order to allow fast **random** access, an index structure may be used.
2. A file may have several indices on different search keys.
3. If the file containing the records is sequentially ordered, the index whose search key specifies the sequential order of the file is the **primary index**, or **clustering index**. Note: The search key of a primary index is usually the primary key, but it is not necessarily so.
4. Indices whose search key specifies an order different from the sequential order of the file are called the **secondary indices**, or **nonclustering indices**.

11.2.1 Primary Index

1. *Index-sequential files*: Files are ordered sequentially on some search key, and a primary index is associated with it.

Dense and Sparse Indices

1. There are Two types of ordered indices:

   Dense Index:

   - An index record appears for **every** search key value in file.
   - This record contains search key value and a pointer to the actual record.

   Sparse Index:

   - Index records are created only for **some** of the records.
   - To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
   - We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.
2. Figures 11.2 and 11.3 show dense and sparse indices for the deposit file.
3. Notice how we would find records for Perryridge branch using both methods. (Do it!)
4. Dense indices are faster in general, but sparse indices require less space and impose less maintenance for insertions and deletions. (Why?)

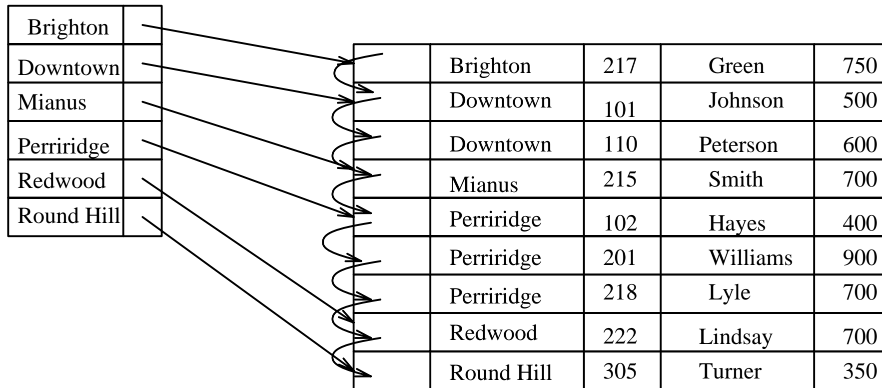


Figure 11.2: Dense index.

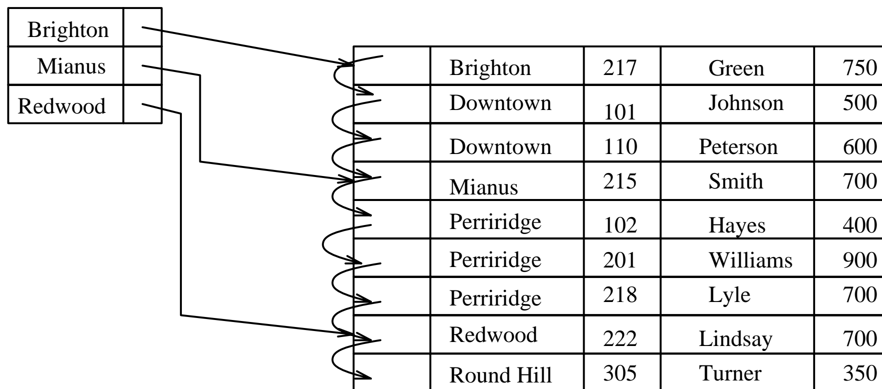


Figure 11.3: Sparse index.

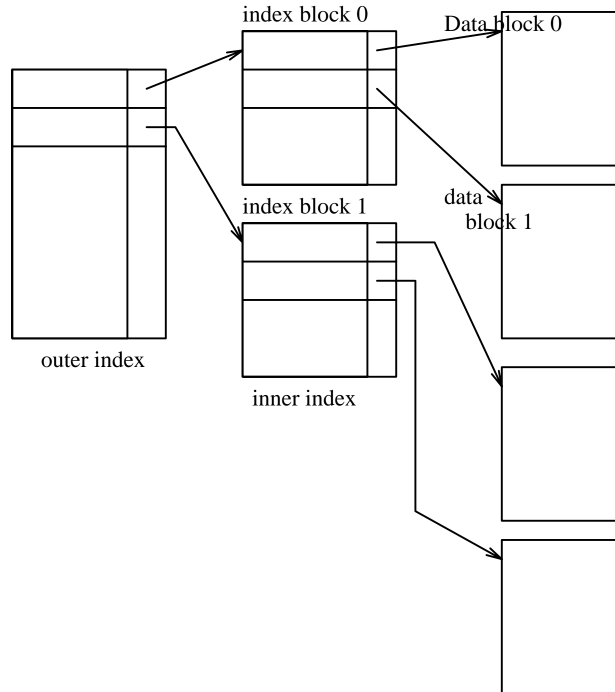


Figure 11.4: Two-level sparse index.

5. A good compromise: to have a sparse index with one entry per block.

   Why is this good?

   - Biggest cost is in bringing a block into main memory.
   - We are guaranteed to have the correct block with this method, unless record is on an overflow block (actually could be **several** blocks).
   - Index size still small.

Multi-Level Indices

1. Even with a sparse index, index size may still grow too large. For 100,000 records, 10 per block, at one index record per block, that's 10,000 index records! Even if we can fit 100 index records per block, this is 100 blocks.
2. If index is too large to be kept in main memory, a search results in several disk reads.
   - If there are no overflow blocks in the index, we can use binary search.
   - This will read as many as $1 + \log_2(b)$ blocks (as many as 7 for our 100 blocks).
   - If index has overflow blocks, then sequential search typically used, reading **all** b index blocks.
3. Solution: Construct a sparse index on the index (Figure 11.4).
4. Use binary search on outer index. Scan **index block** found until correct index record found. Use index record as before - scan block pointed to for desired record.
5. For very large files, additional levels of indexing may be required.
6. Indices must be updated at all levels when insertions or deletions require it.
7. Frequently, each level of index corresponds to a unit of physical storage (e.g. indices at the level of track, cylinder and disk).

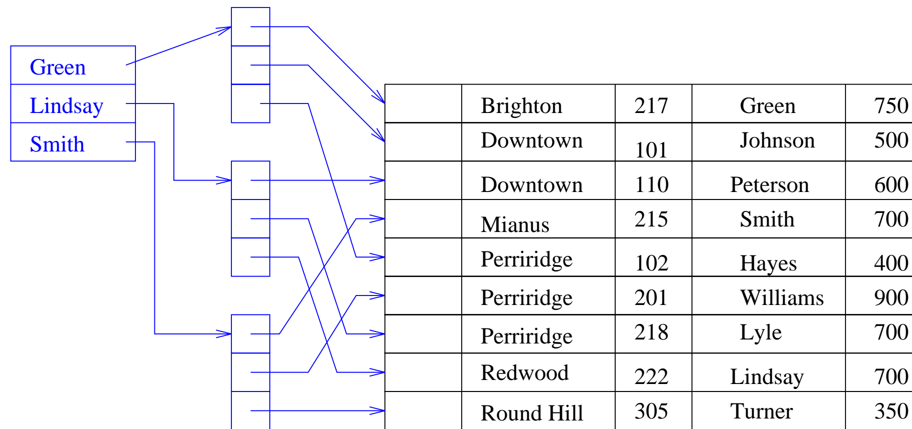


Figure 11.5: Sparse secondary index on *cname*.

Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file.

1. **Deletion:**
   - Find (look up) the record
   - If the last record with a particular search key value, **delete** that search key value from index.
   - For dense indices, this is like deleting a record in a file.
   - For sparse indices, delete a key value by replacing key value's entry in index by next search key value. If that value already has an index entry, delete the entry.
2. **Insertion:**
   - Find place to insert.
   - Dense index: insert search key value if not present.
   - Sparse index: no change unless new block is created. (In this case, the first search key value appearing in the new block is inserted into the index).

11.2.2 Secondary Indices

1. If the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value because the remaining records with the same search-key value could be anywhere in the file. Therefore, a secondary index must contain pointers to all the records.
2. We can use an extra-level of indirection to implement secondary indices on search keys that are not candidate keys. A pointer does not point directly to the file but to a bucket that contains pointers to the file.
   - See Figure 11.5 on secondary key *cname*.
   - To perform a lookup on Peterson, we must read all three records pointed to by entries in bucket 2.
   - Only one entry points to a Peterson record, but three records need to be read.
   - As file is not ordered physically by *cname*, this may take 3 block accesses.
3. Secondary indices must be **dense**, with an index entry for every search-key value, and a pointer to every record in the file.

4. Secondary indices improve the performance of queries on non-primary keys.

5. They also impose serious overhead on database modification: whenever a file is updated, every index must be updated.

6. Designer must decide whether to use secondary indices or not.

11.3 B$^+$-Tree Index Files

1. Primary disadvantage of index-sequential file organization is that performance degrades as the file grows. This can be remedied by costly re-organizations.

2. B$^+$-tree file structure maintains its efficiency despite frequent insertions and deletions. It imposes some acceptable update and space overheads.

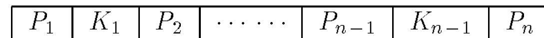
3. A B$^+$-tree index is a *balanced tree* in which every path from the root to a leaf is of the same length.

4. Each nonleaf node in the tree must have between $\lceil n/2 \rceil$ and n children, where n is fixed for a particular tree.

11.3.1 Structure of a B$^+$-Tree

1. A B$^+$-tree index is a *multilevel* index but is structured differently from that of multi-level index sequential files.

2. A typical node (Figure 11.6) contains up to $n-1$ search key values $K_1, K_2, \ldots, K_{n-1}$, and n pointers $P_1, P_2, \ldots, P_n$. Search key values in a node are kept in sorted order.



P_1	K_1	P_2	$\cdots\cdots$	P_{n-1}	K_{n-1}	P_n

Figure 11.6: Typical node of a B+-tree.

3. For leaf nodes, P_i $(i = 1, \ldots, n-1)$ points to either a file record with search key value K_i, or a bucket of pointers to records with that search key value. Bucket structure is used if search key is not a primary key, and file is not sorted in search key order.

   Pointer P_n (nth pointer in the leaf node) is used to chain leaf nodes together in linear order (search key order). This allows efficient **sequential** processing of the file.

   The range of values in each **leaf** do not overlap.

4. Non-leaf nodes form a multilevel index on leaf nodes.

   A non-leaf node may hold up to n pointers and must hold $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fan-out* of the node.

   Consider a node containing m pointers. Pointer P_i $(i = 2, \ldots, m)$ points to a subtree containing search key values $\geq K_{i-1}$ and $< K_i$. Pointer P_m points to a subtree containing search key values $\geq K_{m-1}$. Pointer P_1 points to a subtree containing search key values $< K_1$.

5. Figures 11.7 (textbook Fig. 11.8) and textbook Fig. 11.9 show B$^+$-trees for the *deposit* file with n=3 and n=5.

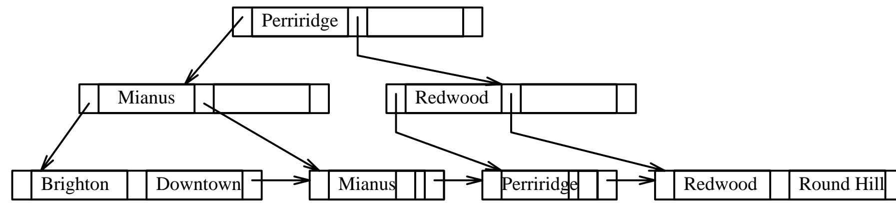


Figure 11.7: B+-tree for *deposit* file with $n = 3$.

11.3.2 Queries on B$^+$-Trees

1. Suppose we want to find all records with a search key value of k.
   - Examine the root node and find the smallest search key value $K_i > k$.
   - Follow pointer P_i to another node.
   - If $k < K_1$ follow pointer P_1.
   - Otherwise, find the appropriate pointer to follow.
   - Continue down through non-leaf nodes, looking for smallest search key value $> k$ and following the corresponding pointer.
   - Eventually we arrive at a leaf node, where pointer will point to the desired record or bucket.

2. In processing a query, we traverse a path from the root to a leaf node. If there are K search key values in the file, this path is no longer than $\log_{\lceil n/2 \rceil}(K)$.

   This means that the path is not long, even in large files. For a $4k$ byte disk block with a search-key size of 12 bytes and a disk pointer of 8 bytes, n is around 200. If $n = 100$, a look-up of 1 million search-key values may take $\log_{50}(1,000,000) = 4$ nodes to be accessed. Since root is in usually in the buffer, so typically it takes only 3 or fewer disk reads.

11.3.3 Updates on B$^+$-Trees

1. **Insertions and Deletions:**

   Insertion and **deletion** are more complicated, as they may require splitting or combining nodes to keep the tree balanced. If splitting or combining are not required, insertion works as follows:
   - Find leaf node where search key value should appear.
   - If value is present, add new record to the bucket.
   - If value is not present, insert value in leaf node (so that search keys are still in order).
   - Create a new bucket and insert the new record.

   If splitting or combining are not required, deletion works as follows:
   - Deletion: Find record to be deleted, and remove it from the bucket.
   - If bucket is now empty, remove search key value from leaf node.

2. **Insertions Causing Splitting:**

   When insertion causes a leaf node to be too large, we **split** that node. In Figure 11.8, assume we wish to insert a record with a *bname* value of "Clearview".
   - There is no room for it in the leaf node where it should appear.
   - We now have n values (the $n-1$ search key values plus the new one we wish to insert).
   - We put the first $\lceil n/2 \rceil$ values in the existing node, and the remainder into a new node.

- Figure 11.10 shows the result.
- The new node must be inserted into the B$^+$-tree.
- We also need to update search key values for the parent (or higher) nodes of the split leaf node. (Except if the new node is the leftmost one)
- Order must be preserved among the search key values in each node.
- If the parent was already full, it will have to be split.
- When a non-leaf node is split, the children are divided among the two new nodes.
- In the worst case, splits may be required all the way up to the root. (If the root is split, the tree becomes one level deeper.)
- **Note:** when we start a B$^+$-tree, we begin with a single node that is both the root and a single leaf. When it gets full and another insertion occurs, we split it into two leaf nodes, requiring a new root.

3. **Deletions Causing Combining:**

   Deleting records may cause tree nodes to contain too few pointers. Then we must combine nodes.

   - If we wish to delete "Downtown" from the B$^+$-tree of Figure 11.11, this occurs.
   - In this case, the leaf node is empty and must be deleted.
   - If we wish to delete "Perryridge" from the B$^+$-tree of Figure 11.11, the parent is left with only one pointer, and must be coalesced with a sibling node.
   - Sometimes higher-level nodes must also be coalesced.
   - If the root becomes empty as a result, the tree is one level less deep (Figure 11.13).
   - Sometimes the pointers must be redistributed to keep the tree balanced.
   - Deleting "Perryridge" from Figure 11.11 produces Figure 11.14.

4. To summarize:

   - Insertion and deletion are complicated, but require relatively few operations.
   - Number of operations required for insertion and deletion is proportional to logarithm of number of search keys.
   - B$^+$-trees are fast as index structures for database.

11.3.4 B$^+$-Tree File Organization

1. The B$^+$-tree structure is used not only as an index but also as an organizer for records into a file.
2. In a B$^+$-tree file organization, the leaf nodes of the tree store records instead of storing pointers to records, as shown in Fig. 11.17.
3. Since records are usually larger than pointers, the maximum number of records that can be stored in a leaf node is less than the maximum number of pointers in a nonleaf node.
4. However, the leaf node are still required to be at least half full.
5. Insertion and deletion from a B$^+$-tree file organization are handled in the same way as that in a B$^+$-tree index.
6. When a B$^+$-tree is used for file organization, space utilization is particularly important. We can improve the space utilization by involving more sibling nodes in redistribution during splits and merges.
7. In general, if m nodes are involved in redistribution, each node can be guaranteed to contain at least $\lceil (m-1)n/m \rceil$ entries. However, the cost of update becomes higher as more siblings are involved in redistribution.

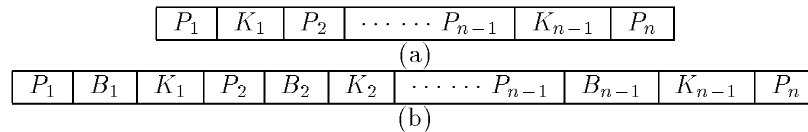


Figure 11.8: Leaf and nonleaf node of a B-tree.

11.4 B-Tree Index Files

1. B-tree indices are similar to B$^+$-tree indices.

   - Difference is that B-tree eliminates the redundant storage of search key values.
   - In B$^+$-tree of Figure 11.11, some search key values appear twice.
   - A corresponding B-tree of Figure 11.18 allows search key values to appear only once.
   - Thus we can store the index in less space.

2. **Advantages:**

   - Lack of redundant storage (but only marginally different).
   - Some searches are faster (key may be in non-leaf node).

3. **Disadvantages:**

   - Leaf and non-leaf nodes are of different size (complicates storage)
   - Deletion may occur in a non-leaf node (more complicated)

   Generally, the structural simplicity of B$^+$-tree is preferred.

11.5 Static Hashing

1. Index schemes force us to traverse an index structure. Hashing avoids this.

11.5.1 Hash File Organization

1. **Hashing** involves computing the address of a data item by computing a function on the search key value.

2. A **hash function h** is a function from the set of all search key values K to the set of all bucket addresses B.

   - We choose a number of buckets to correspond to the number of search key values we will have stored in the database.
   - To perform a lookup on a search key value K_i, we compute $h(K_i)$, and search the bucket with that address.
   - If two search keys i and j map to the same address, because $h(K_i) = h(K_j)$, then the bucket at the address obtained will contain records with both search key values.
   - In this case we will have to check the search key value of every record in the bucket to get the ones we want.
   - Insertion and deletion are simple.

Hash Functions

1. A good hash function gives an average-case lookup that is a small constant, independent of the number of search keys.
2. We hope records are distributed uniformly among the buckets.
3. The worst hash function maps all keys to the same bucket.
4. The best hash function maps all keys to distinct addresses.
5. Ideally, distribution of keys to addresses is uniform and random.
6. Suppose we have 26 buckets, and map names beginning with ith letter of the alphabet to the ith bucket.
   - Problem: this does not give uniform distribution.
   - Many more names will be mapped to "A" than to "X".
   - Typical hash functions perform some operation on the internal binary machine representations of characters in a key.
   - For example, compute the sum, modulo # of buckets, of the binary representations of characters of the search key.
   - See Figure 11.18, using this method for 10 buckets (assuming the ith character in the alphabet is represented by integer i).

Handling of bucket overflows

1. **Open** hashing occurs where records are stored in different buckets. Compute the hash function and search the corresponding bucket to find a record.
2. **Closed** hashing occurs where all records are stored in **one** bucket. Hash function computes addresses within that bucket. (Deletions are difficult.) Not used much in database applications.
3. **Drawback to our approach:** Hash function must be chosen at implementation time.
   - Number of buckets is fixed, but the database may grow.
   - If number is too large, we waste space.
   - If number is too small, we get too many "collisions", resulting in records of many search key values being in the same bucket.
   - Choosing the number to be twice the number of search key values in the file gives a good space/performance tradeoff.

11.5.2 Hash Indices

1. A hash index organizes the search keys with their associated pointers into a hash file structure.
2. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).
3. Strictly speaking, hash indices are only secondary index structures, since if a file itself is organized using hashing, there is no need for a separate hash index structure on it.

11.6 Dynamic Hashing

1. As the database grows over time, we have three options:
   - Choose hash function based on current file size. Get performance degradation as file grows.
   - Choose hash function based on anticipated file size. Space is wasted initially.

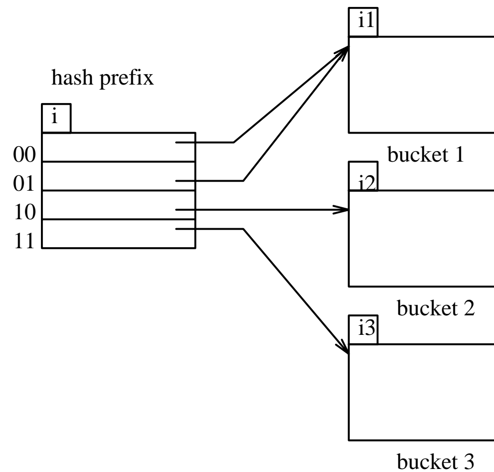


Figure 11.9: General extendable hash structure.

- Periodically re-organize hash structure as file grows. Requires selecting new hash function, recomputing all addresses and generating new bucket assignments. Costly, and shuts down database.

2. Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinking of the database. These are called **dynamic hash functions**.

   - **Extendable hashing** is one form of dynamic hashing.
   - Extendable hashing splits and coalesces buckets as database size changes.
   - This imposes some performance overhead, but space efficiency is maintained.
   - As reorganization is on one bucket at a time, overhead is acceptably low.

3. How does it work?

   - We choose a hash function that is uniform and random that generates values over a relatively large range.
   - Range is b-bit binary integers (typically b=32).
   - 2^{32} is over 4 billion, so we don't generate that many buckets!
   - Instead we create buckets on demand, and do not use all b bits of the hash initially.
   - At any point we use i bits where $0 \leq i \leq b$.
   - The i bits are used as an offset into a table of bucket addresses.
   - Value of i grows and shrinks with the database.
   - Figure 11.19 shows an extendable hash structure.
   - Note that the i appearing over the bucket address table tells how many bits are required to determine the correct bucket.
   - It may be the case that several entries point to the same bucket.
   - All such entries will have a common hash prefix, but the length of this prefix may be less than i.
   - So we give each bucket an integer giving the length of the common hash prefix.
   - This is shown in Figure 11.9 (textbook 11.19) as i_j.
   - Number of bucket entries pointing to bucket j is then $2^{(i-i_j)}$.

4. To find the bucket containing search key value K_l:

   - Compute $h(K_l)$.

- Take the first i high order bits of $h(K_l)$.
- Look at the corresponding table entry for this i-bit string.
- Follow the bucket pointer in the table entry.

5. We now look at insertions in an extendable hashing scheme.
   - Follow the same procedure for lookup, ending up in some bucket j.
   - If there is room in the bucket, insert information and insert record in the file.
   - If the bucket is full, we must split the bucket, and redistribute the records.
   - If bucket is split we may need to increase the number of bits we use in the hash.
6. Two cases exist:
   1. If $i = i_j$, then only one entry in the bucket address table points to bucket j.
      - Then we need to increase the size of the bucket address table so that we can include pointers to the two buckets that result from splitting bucket j.
      - We increment i by one, thus considering more of the hash, and doubling the size of the bucket address table.
      - Each entry is replaced by two entries, each containing original value.
      - Now two entries in bucket address table point to bucket j.
      - We allocate a new bucket z, and set the second pointer to point to z.
      - Set i_j and i_z to i.
      - Rehash all records in bucket j which are put in either j or z.
      - Now insert new record.
      - It is remotely possible, but unlikely, that the new hash will still put all of the records in one bucket.
      - If so, split again and increment i again.
   2. If $i > i_j$, then more than one entry in the bucket address table points to bucket j.
      - Then we can split bucket j without increasing the size of the bucket address table (why?).
      - Note that all entries that point to bucket j correspond to hash prefixes that have the same value on the leftmost i_j bits.
      - We allocate a new bucket z, and set i_j and i_z to the original i_j value plus 1.
      - Now adjust entries in the bucket address table that previously pointed to bucket j.
      - Leave the first half pointing to bucket j, and make the rest point to bucket z.
      - Rehash each record in bucket j as before.
      - Reattempt new insert.
7. Note that in both cases we only need to rehash records in bucket j.
8. Deletion of records is similar. Buckets may have to be coalesced, and bucket address table may have to be halved.
9. Insertion is illustrated for the example *deposit* file of Figure 11.20.
   - 32-bit hash values on *bname* are shown in Figure 11.21.
   - An initial empty hash structure is shown in Figure 11.22.
   - We insert records one by one.
   - We (unrealistically) assume that a bucket can only hold 2 records, in order to illustrate both situations described.

- As we insert the Perryridge and Round Hill records, this first bucket becomes full.
- When we insert the next record (Downtown), we must split the bucket.
- Since $i = i_0$, we need to increase the number of bits we use from the hash.
- We now use 1 bit, allowing us $2^1 = 2$ buckets.
- This makes us double the size of the bucket address table to two entries.
- We split the bucket, placing the records whose search key hash begins with 1 in the new bucket, and those with a 0 in the old bucket (Figure 11.23).
- Next we attempt to insert the Redwood record, and find it hashes to 1.
- That bucket is full, and $i = i_1$.
- So we must split that bucket, increasing the number of bits we must use to 2.
- This necessitates doubling the bucket address table again to four entries (Figure 11.24).
- We rehash the entries in the old bucket.
- We continue on for the deposit records of Figure 11.20, obtaining the extendable hash structure of Figure 11.25.

10. **Advantages:**
    - Extendable hashing provides performance that does not degrade as the file grows.
    - Minimal space overhead - no buckets need be reserved for future use. Bucket address table only contains one pointer for each hash value of current prefix length.
11. **Disadvantages:**
    - Extra level of indirection in the bucket address table
    - Added complexity
12. **Summary:** A highly attractive technique, provided we accept added complexity.

11.7 Comparison of Indexing and Hashing

1. To make a wise choice between the methods seen, database designer must consider the following issues:
    - Is the cost of periodic re-organization of index or hash structure acceptable?
    - What is the relative frequence of insertion and deletion?
    - Is it desirable to optimize average access time at the expense of increasing worst-case access time?
    - What types of queries are users likely to pose?
2. The last issue is critical to the choice between indexing and hashing. If most queries are of the form

    select $A_1, A_2, \ldots, A_n$
    from r
    where $A_i = c$

    then to process this query the system will perform a lookup on an index or hash structure for attribute A_i with value c.
3. For these sorts of queries a hashing scheme is preferable.
    - Index lookup takes time proportional to log of number of values in R for A_i.
    - Hash structure provides lookup average time that is a small constant (independent of database size).
4. However, the worst-case favors indexing:
    - Hash worst-case gives time proportional to the **number** of values in R for A_i.

- Index worst case still **log** of number of values in R.

5. Index methods are preferable where a **range** of values is specified in the query, e.g.

   select $A_1, A_2, \ldots, A_n$
   from r
   where $A_i \leq c_2$ **and** $A_i \geq c_1$

   This query finds records with A_i values in the **range** from c_1 to c_2.

6. - Using an index structure, we can find the bucket for value c_1, and then follow the pointer chain to read the next buckets in alphabetic (or numeric) order until we find c_2.
   - If we have a hash structure instead of an index, we can find a bucket for c_1 easily, but it is not easy to find the "next bucket".
   - A good hash function assigns values **randomly** to buckets.
   - Also, each bucket may be assigned many search key values, so we cannot chain them together.
   - To support range queries using a hash structure, we need a hash function that preserves order.
   - For example, if K_1 and K_2 are search key values and $K_1 < K_2$ then $h(K_1) < h(K_2)$.
   - Such a function would ensure that buckets are in key order.
   - Order-preserving hash functions that also provide randomness and uniformity are extremely difficult to find.
   - Thus most systems use indexing in preference to hashing unless it is known in advance that range queries will be infrequent.

11.8 Index Definition in SQL

1. Some SQL implementations includes data definition commands to create and drop indices. The IBM SAA-SQL commands are

   - An index is created by

     create index <index-name>
     on r (<attribute-list>)

   - The attribute list is the list of attributes in relation r that form the search key for the index.
   - To create an index on *bname* for the *branch* relation:

     create index *b-index*
     on *branch* (*bname*)

   - If the search key is a candidate key, we add the word **unique** to the definition:

     create unique index *b-index*
     on *branch* (*bname*)

   - If *bname* is not a candidate key, an error message will appear.
   - If the index creation succeeds, any attempt to insert a tuple violating this requirement will fail.
   - The **unique** keyword is redundant if primary keys have been defined with integrity constraints already.

2. To remove an index, the command is

   drop index <index-name>

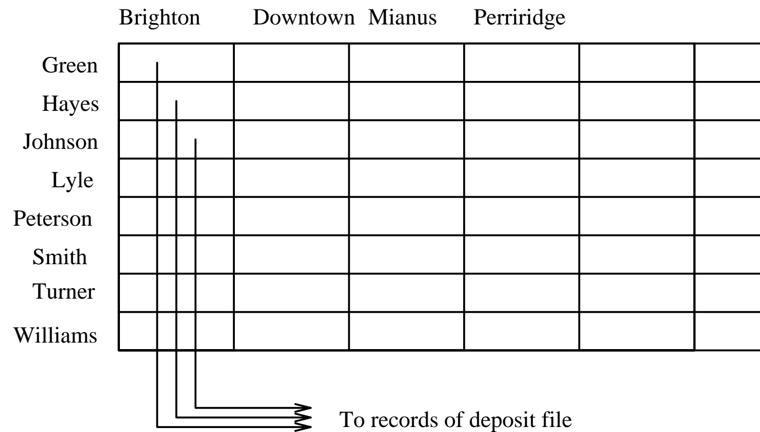


Figure 11.10: Grid structure for *deposit* file.

11.9 Multiple-Key Access

1. For some queries, it is advantageous to use multiple indices if they exist.
2. If there are two indices on *deposit*, one on *bname* and one on *cname*, then suppose we have a query like

   select *balance*
   from *deposit*
   where *bname* = "Perryridge" **and** *balance* = 1000
3. There are 3 possible strategies to process this query:
   - Use the index on *bname* to find all records pertaining to Perryridge branch. Examine them to see if *balance* = 1000
   - Use the index on *balance* to find all records pertaining to Williams. Examine them to see if *bname* = "Perryridge".
   - Use index on *bname* to find pointers to records pertaining to Perryridge branch. Use index on *balance* to find pointers to records pertaining to 1000. Take the **intersection** of these two sets of pointers.
4. The third strategy takes advantage of the existence of multiple indices. This may still not work well if
   - There are a large number of Perryridge records **AND**
   - There are a large number of 1000 records **AND**
   - Only a small number of records pertain to both Perryridge and 1000.
5. To speed up multiple search key queries special structures can be maintained.

11.9.1 Grid File

1. A grid structure for queries on two search keys is a 2-dimensional grid, or array, indexed by values for the search keys. Figure 11.10 (textbook 11.31) shows part of a grid structure for the *deposit* file.
2. A particular entry in the array contains pointers to all records with the specified search key values.
   - No special computations need to be done
   - Only the right records are accessed
   - Can also be used for single search key queries (one column or row)
   - Easy to extend to queries on n search keys – construct an n-dimensional array.

- Significant improvement in processing time for multiple-key queries.
- Imposes space overhead.
- Performance overhead on insertion and deletion.

11.9.2 Partitioned Hashing

1. - An alternative approach to multiple-key queries.
   - To construct a structure for queries on deposit involving *bname* and *cname*, we construct a hash structure for the key (*cname, bname*).
   - We split this hash function into two parts, one for each part of the key.
   - The first part depends only on the *cname* value.
   - The second part depends only on the *bname* value.
   - Figure 11.32 shows a sample partitioned hash function.
   - Note that pairs with the same *cname* or *bname* value will have 3 bits the same in the appropriate position.
   - To find the balance in all of Williams' accounts at the Perryridge branch, we compute h(Williams, Perryridge) and access the hash structure.
2. The same hash structure can be used to answer a query on **one** of the search keys:
   - Compute part of partitioned hash.
   - Access hash structure and scan buckets for which that part of the hash coincides.
   - Text doesn't say so, but the hash structure must have some grid-like form imposed on it to enable searching the structure based on only some part of the hash.
3. Partitioned hashing can also be extended to n-key search.