Indexing and Hashing

Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

  select account_number

  from account

  where branch_name = “Perryridge” and balance = 1000
- Possible strategies for processing query using indices on single attributes:
  - 1.Use index on to find accounts with balances of $1000; test branch_name = “Perryridge”.
  - 2.Use index on to find accounts with balances of $1000; test branch_name = “Perryridge”.
  - 3.Use to find pointers to all records pertaining to the . Similarly use index on balance. Take of both sets of pointers obtained.

Indices on Multiple Keys

- are search keys containing more than one attribute
  - E.g. (branch_name, balance)
- ordering: (a1, a2) < (b1, b2) if either
  - a1 < a2, or
  - a1=a2 and a2 < b2

Indices on Multiple Attributes

Suppose we have an index on combined search-key (*branch_name, balance*).

- With the **where** clause
  where *branch_name* = "Perryridge" **and** *balance* = 1000
  the index on (*branch_name, balance*) can be used to fetch records that satisfy conditions.
  - Using separate indices is — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also handle
  where *branch_name* = "Perryridge" **and** *balance* < 1000
- But efficiently handle
  where *branch_name* < "Perryridge" **and** *balance* = 1000
  - May fetch many records that satisfy the first but not the second condition

Non-Unique Search Keys

- Alternatives:
  - on separate block (bad idea)
  - List of with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a
    - Extra storage for keys
    - Simpler code for insertion/deletion
    - Widely used

Other Issues

- Covering indices
  - Add extra attributes to index so (some) queries can avoid fetching the actual records
    - Particularly useful for secondary indices
  - Can store extra attributes only at leaf
- Record and secondary
  - If a record moves, all secondary indices that store record pointers have to be
  - Node splits in B+-tree file organizations become
  - Solution: use key instead of pointer in secondary index
    - Extra traversal of primary index to locate record
      - Higher cost for queries, but node splits are cheap
    - Add record-id if key is non-unique

Static Hashing

- A is a unit of storage containing one or more records (a is typically a disk block).
- In a file organization we obtain the bucket of a record directly from its search-key value using a
- h is a function from the set of all search-key values K to the set of all B.
- Hash function is used to records for access, as well as .
- Records with different search-key values may be to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

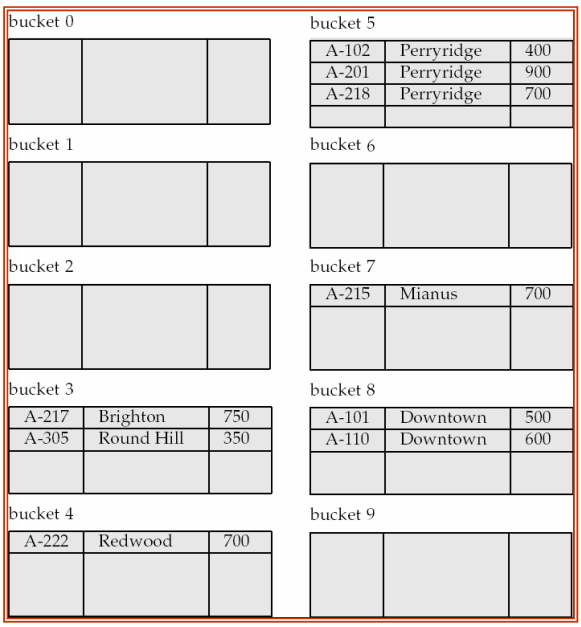
Example of Hash File Organization

Hash file organization of *account* file, using *branch_name* as key
(See figure in next slide.)

- There are 10 buckets,
- The of the *i*th character is assumed to be the integer *i*.
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Perryridge) = 5
  - h(Round Hill) = 3 h(Brighton) = 3

Example of Hash File Organization

Hash file organization of *account* file, using *branch_name* as key (see previous slide for details).



Hash Functions

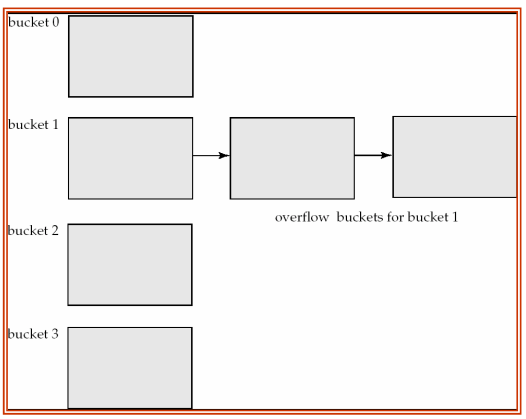
- hash function maps search-key values to the ; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is , i.e., each bucket is assigned the from the set of all possible values.
- Ideal hash function is , so each bucket will have the assigned to it irrespective of the actual distribution of search-key values in the file.
- Typical hash functions perform on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.

Handling of Bucket Overflows

- Bucket can occur because of
  - buckets
  - in distribution of records. This can occur due to two reasons:
    - multiple records have
    - chosen hash function produces of key values
- Although the probability of bucket overflow can be , it cannot be ; it is handled by using .

Handling of Bucket Overflows

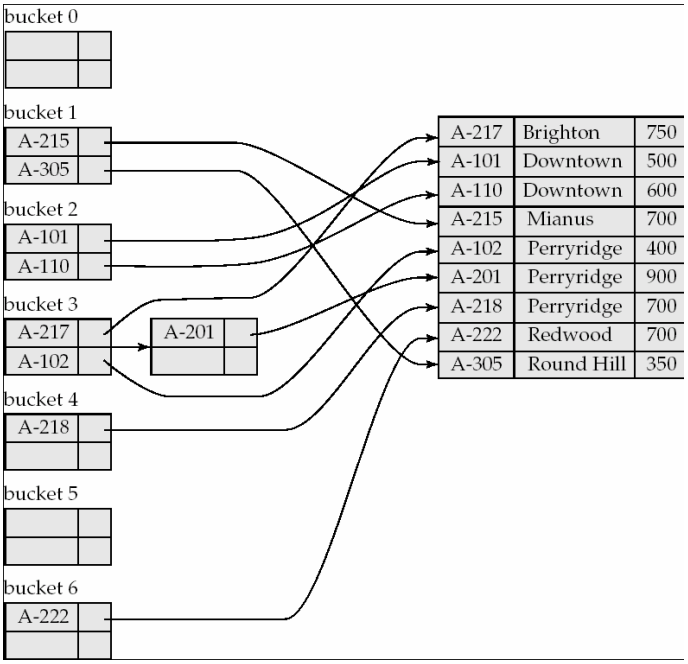
- – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called .
  - An alternative, called , which does not use overflow buckets, is not suitable for database applications.



Hash Indices

- Hashing can be used not only for , but also for creation.
- A organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always
  - if the file itself is organized using , a separate primary on it using the same search-key is unnecessary.
  - However, we use the term to refer to both secondary index structures and hash organized files.

Example of Hash Index



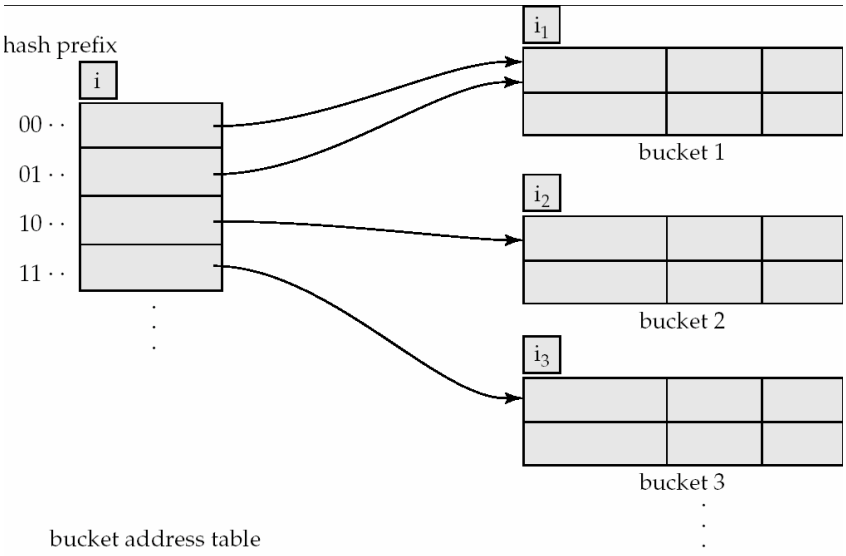
Deficiencies of Static Hashing

- In hashing, function h maps search-key values to a set of B of bucket addresses.
  - Databases . If initial number of buckets is too small, performance will due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be initially.
  - If database , again space will be .
  - One option is periodic of the file with a new hash function, but it is very .
- These problems can be by using techniques that allow the number of buckets to be modified .

Dynamic Hashing

- Good for database that and in size
- Allows the hash function to be modified
- – one form of dynamic hashing
  - Hash function generates values over a range — typically b-bit integers, with b = 32.
  - At any time use only a of the hash function to index into a table of bucket addresses.
  - Let the length of the be i bits, $0 \le i \le 32$.
  - Bucket Initially i = 0
  - Value of grows and shrinks as the size of the database grows and shrinks.
  - in the bucket address table may point to a bucket.
  - Thus, actual number of is < 2i
    - The number of buckets also changes due to and of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

Use of Extendable Hash Structure

- Each stores a value ij; all the entries that point to the same bucket have the bits.
- To locate the bucket containing search-key Kj:
  - 1.Compute h(Kj) = X
  - 2.Use the as a displacement into bucket address table, and follow the to appropriate bucket
- To a record with search-key value Kj
  - 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 (next slide.)
    - Overflow buckets used instead in some cases

Updates in Extendable Hash Structure

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

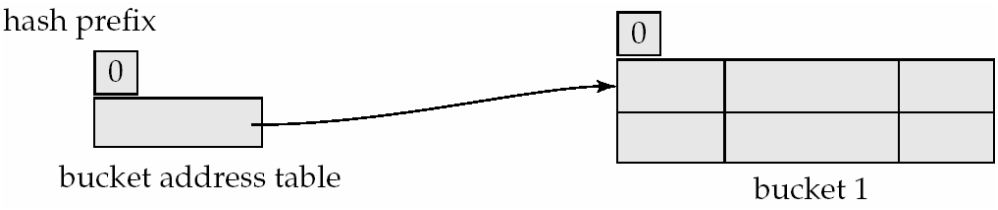
- If i > ij (more than one pointer to bucket j)
  - allocate a , and set ij and iz to the old ij -+ 1.
  - make the second half of the bucket address table entries pointing to j to point to z
  - remove and each record in bucket j.
  - new bucket for Kj and insert record in the bucket (further splitting is required if the bucket is still full)
- If i = ij (only one pointer to bucket j)
  - i and double the size of the bucket address table.
  - replace each entry in the table by that point to the same bucket.
  - recompute new bucket address table entry for Kj
    Now i > ij so use the first case above.

Updates in Extendable Hash Structure

- When inserting a value, if the bucket is full after several splits (that is, i reaches some limit b) create an instead of splitting bucket entry table further.
- To delete a key value,
  - locate it in its bucket and remove it.
  - The (with appropriate updates to the bucket address table).
  - of buckets can be done (can coalesce only with a "buddy" bucket having same value of ij and same ij –1 prefix, if it is present)
  - bucket address table size is also possible
    - Note: decreasing bucket address table size is an operation and should be done only if number of buckets becomes much than the size of the table

Use of Extendable Hash Structure: Example

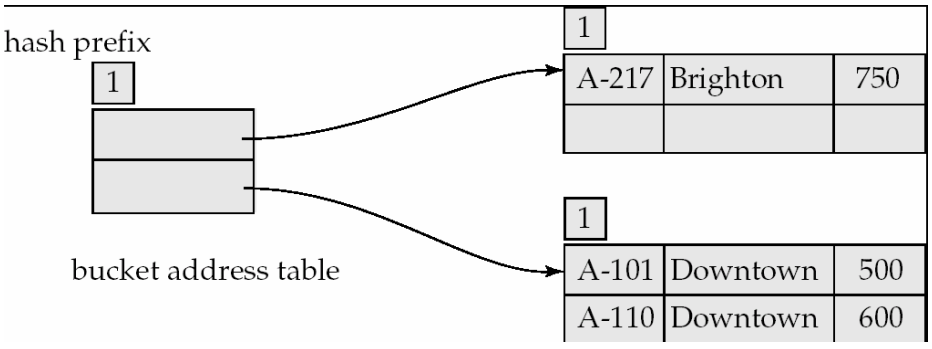
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Initial Hash structure, bucket size = 2

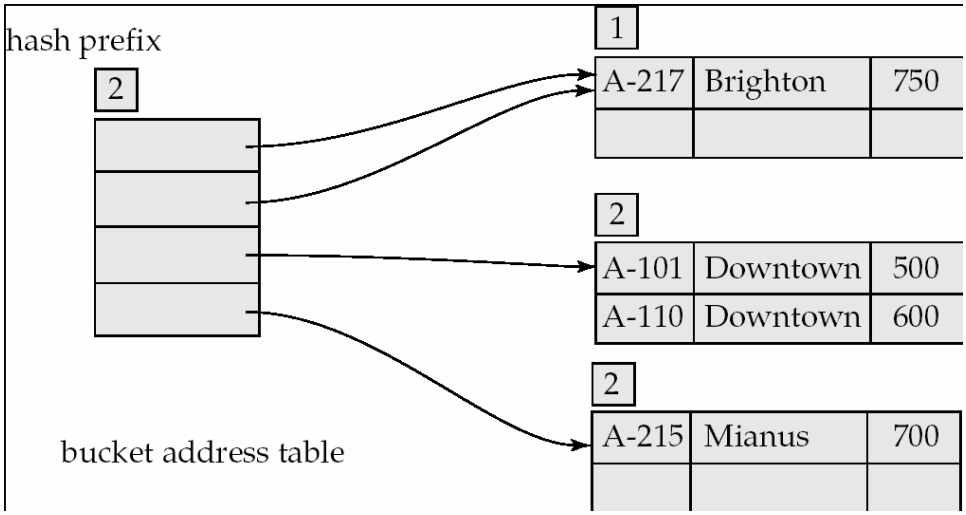
Example

- Hash structure after insertion of one Brighton and two Downtown records

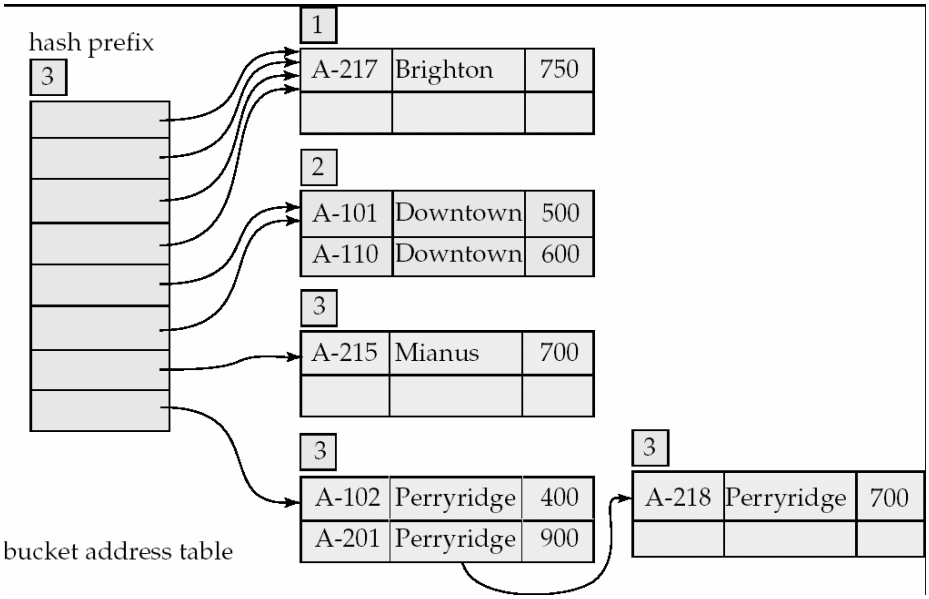


Example

- Hash structure after insertion of Mianus record



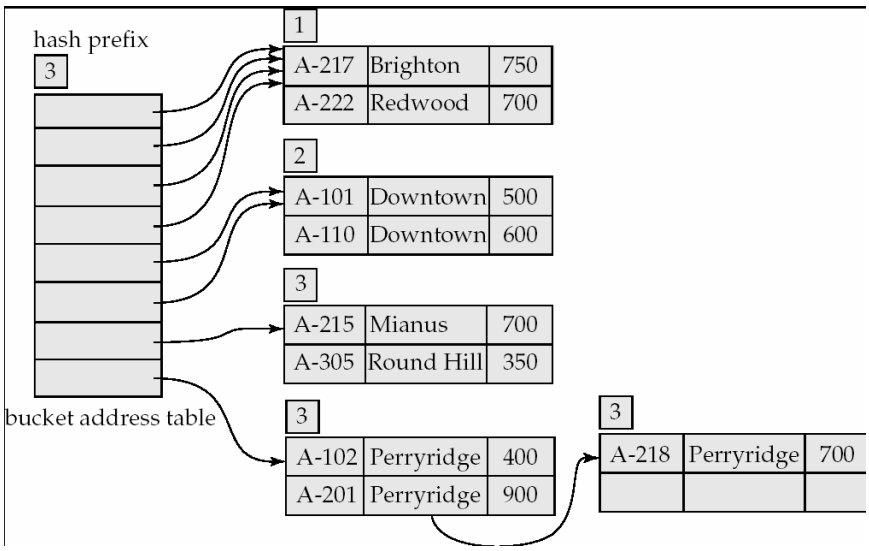
Example



Hash structure after insertion of three Perryridge records

Example

- Hash structure after insertion of Redwood and Round Hill records



Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not with growth of file
  - space overhead
- Disadvantages of extendable hashing
  - Extra level of to find desired record
  - Bucket address table may itself become very
    - Need a to locate desired record in the structure!
  - Changing size of bucket address table is an operation
- is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows

Comparison of 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 to be preferred

Bitmap Indices

- are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered 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 that take on a relatively small number of
  - 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

Bitmap Indices

- In its simplest form a bitmap index on an attribute has a bitmap for of the attribute
  - Bitmap bits as
  - 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

record number	*name*	*gender*	*address*	*income_level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices

- Bitmap indices are useful for queries on
  - not particularly useful for
- Queries are answered using bitmap operations
  - (and)
  - (or)
  - (not)

Bitmap Indices

- Each operation takes of the same size and applies the operation on bits to get the result bitmap
  - E.g. 100110 AND 110011 = 100010
    100110 OR 110011 = 110111
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required .
    - number of matching tuples is even faster

Bitmap Indices

- Bitmap indices generally very compared with 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
- needs to be handled properly
  - Existence bitmap to if there is a valid record at a record location
  - Needed for
    - not(A=v): (NOT bitmap-A-v) AND ExistenceBitmap
- Should keep for all values, even
  - To correctly handle SQL null semantics for NOT(A=v):
    - intersect above result with (NOT bitmap-A-Null)

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 anded with just 31,250 instruction
- 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 to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of 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

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name> (<attribute-list>)

  E.g.: **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 is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>