Database Engineering

Unit 4

Data Storage and Indexing

Prof. Vivek V. Kheradkar

Introduction

- Physical level, logical level, view level
- Storage devices
- Files

Overview of Physical Storage Media

- **Cache**-
- The cache is the fastest and most costly form of storage.
- Cache memory is small;
- its use is managed by the computer system

hardware.

- The cache memory is similar to the main memory
- but is a smaller bin that performs faster.
- The cache memory performs faster by accessing information in fewer clock cycles.
- There are two types of cache memory present in the majority of systems shipped.
- The Level 1 (L1) cache is in the Intel Pentium processor,
- the Level 2 (L2) cache memory is optional and found on the motherboard of most Intel Pentium processor-based systems.

- **Main memory**-
- The storage medium used for data that are available to be operated on is main memory.
- RAM

- **Flash memory**.
- Also known as electrically erasable programmable read-only memory (EEPROM),
- flash memory differs from main memory in that data survive power failure.

- **Magnetic-disk storage**
- The primary medium for the long-term on-line storage of data is the magnetic disk.

- **Optical storage.**
- The most popular forms of optical storage are the compact disk (CD),
- which can hold about 700 megabytes of data,
- and the digital versatile disk (DVD) which can hold

4.7 GB of data

- **Tape storage**
- Tape storage is used primarily for backup and archival data.
- Although magnetic tape is much cheaper than disks, access to data is much slower,
- because the tape must be accessed sequentially

from the beginning.

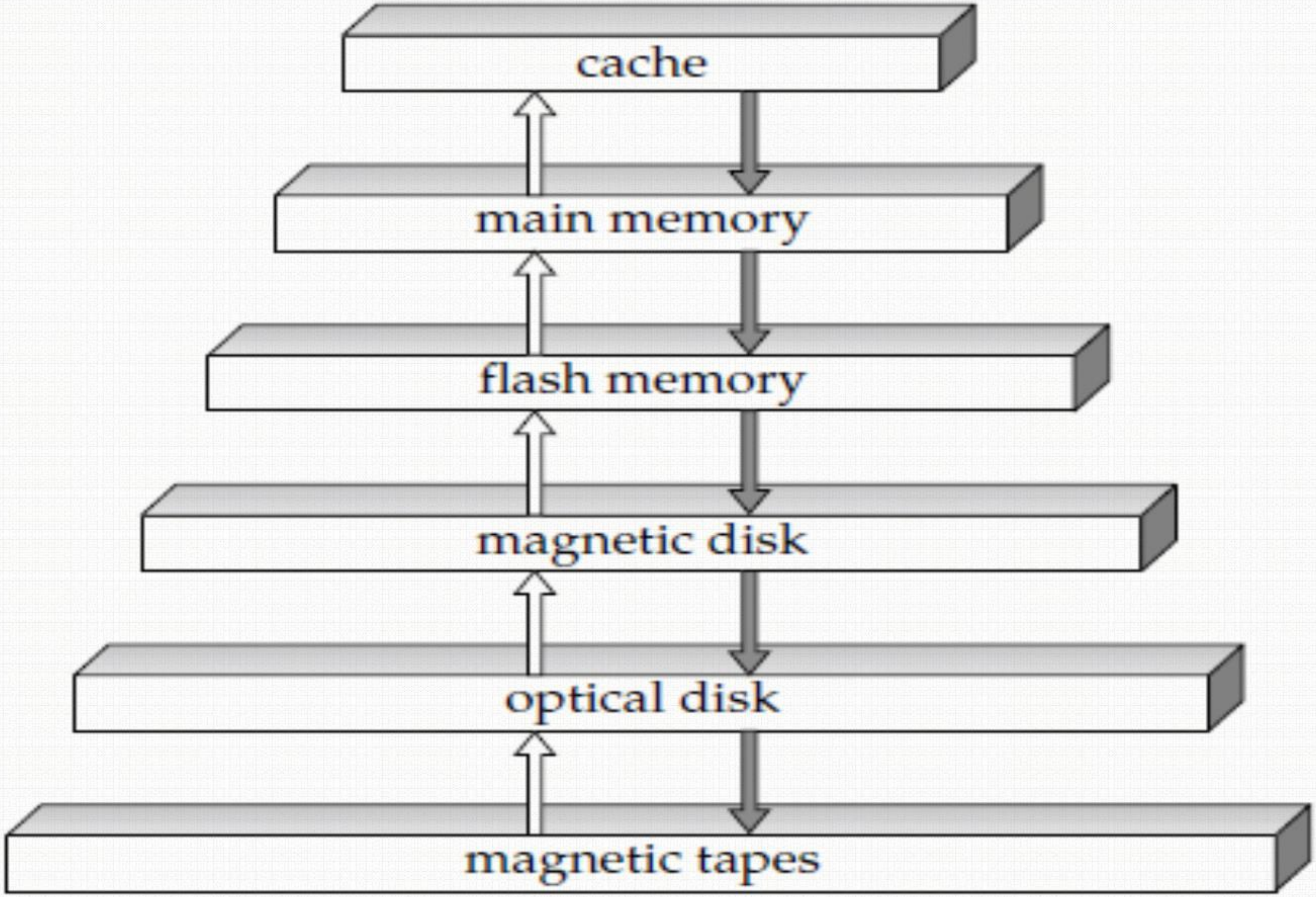


Figure 11.1 Storage-device hierarchy.

Magnetic Disks

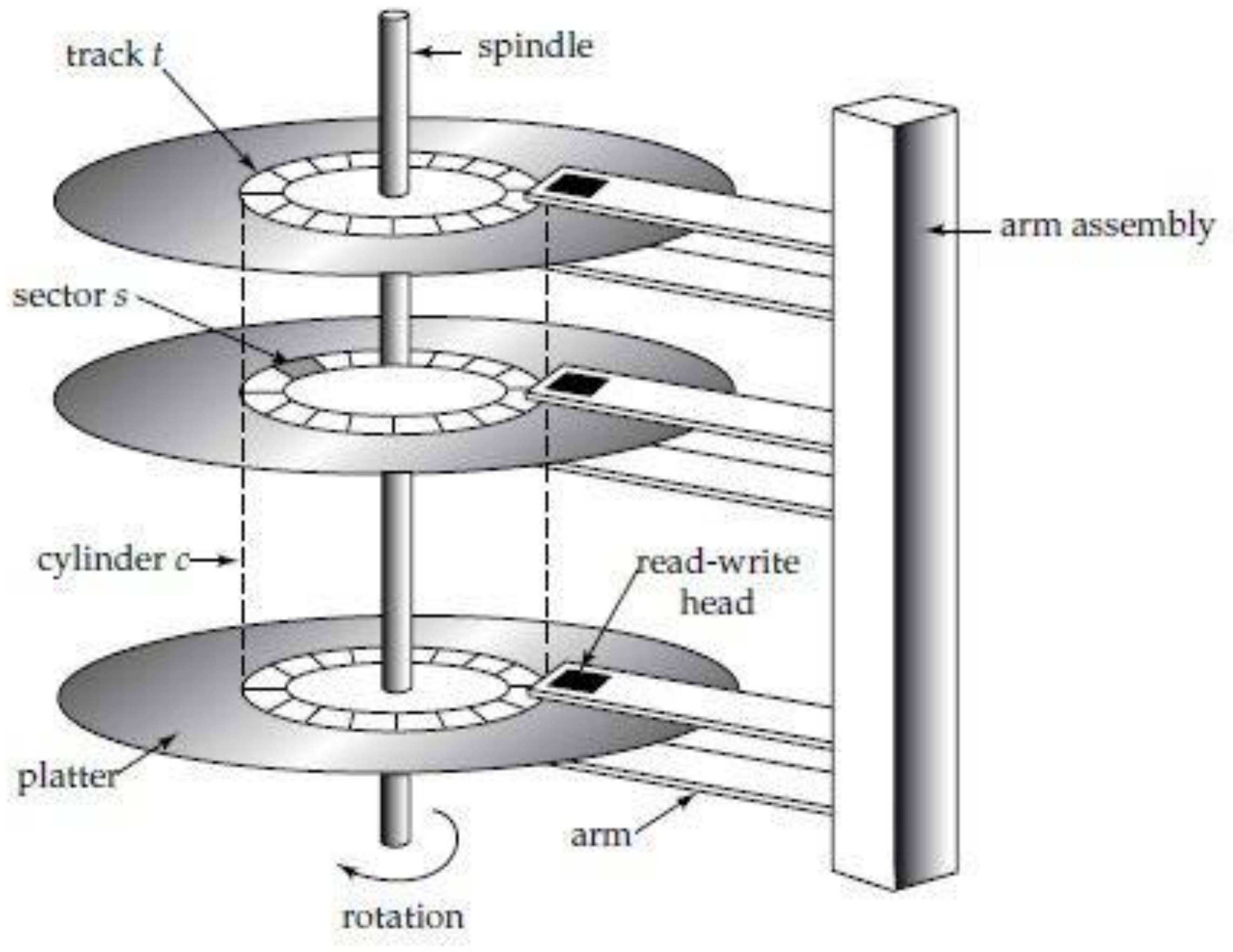


Figure 11.2 Moving-head disk mechanism.

Performance Measures of Disks

- Seek Time
- it is the time taken to reposition and settle the arm and the head over the correct track.
- The lower the seek time, the faster the I/O operation.

- Rotational Latency
- To access data,
- the actuator arm moves the R/W head over the platter to a particular track
- while the platter spins to position the requested

sector under the R/W head.

- The time taken by the platter to rotate and position the data under the R/W head is called rotational latency.
- This latency depends on the rotation speed of the spindle and is measured in milliseconds.

- Data Transfer Rate
- The data transfer rate refers to the average amount of data per unit time that the drive can deliver

RAID
Redundant Arrays of Independent Disks

- RAID or Redundant Array of Independent Disks, is a technology to connect multiple secondary storage devices and use them as a single storage media.

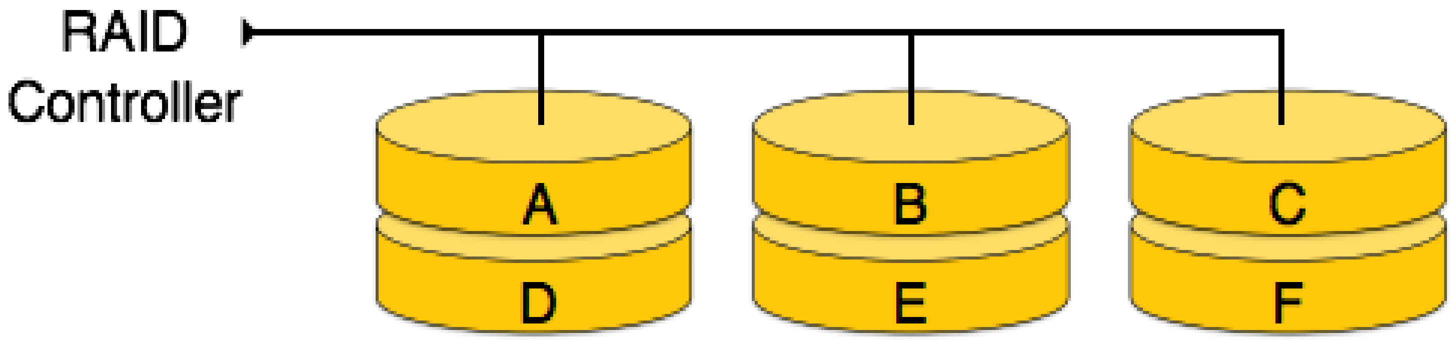
- RAID consists of an array of disks in which multiple disks are connected together to achieve different goals. RAID levels define the use of disk arrays.

- RAID levels are defined on the basis of

  Striping mirroring parity

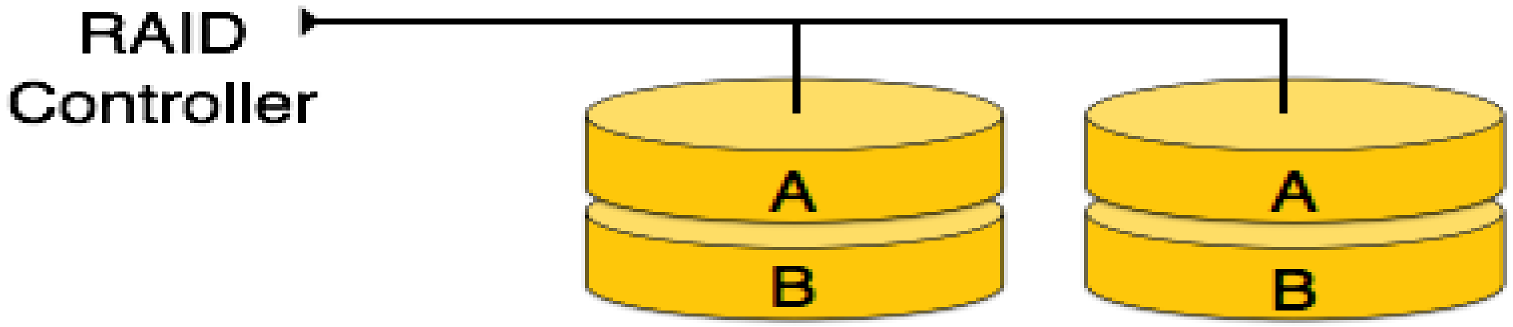
RAID 0

- In this level, a striped array of disks is implemented.
- The data is broken down into blocks and the blocks are distributed among disks. Each disk receives a block of dat to write/read in parallel.
- It enhances the speed and performance of the storage device. There is no parity and backup in Level 0.



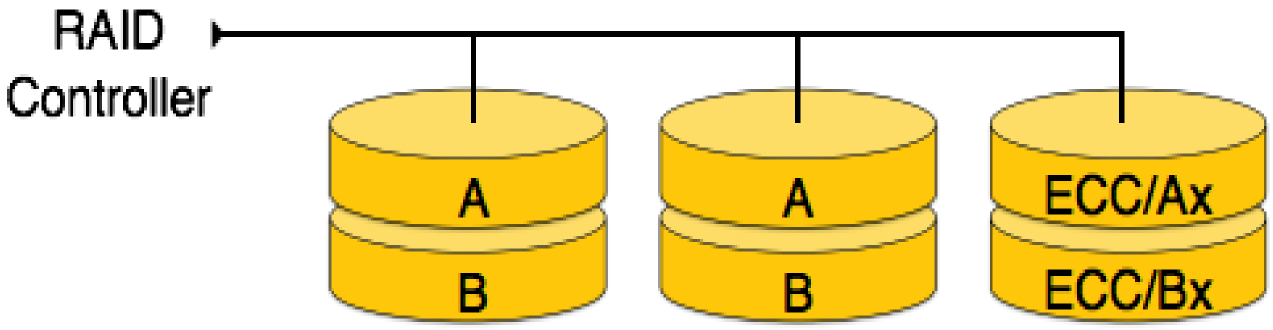
RAID 1

- RAID 1 uses mirroring techniques.
- When data is sent to a RAID controller, it sends a copy of data to all the disks in the array.
- RAID level 1 is also called mirroring and provides 100% redundancy in case of a failure.



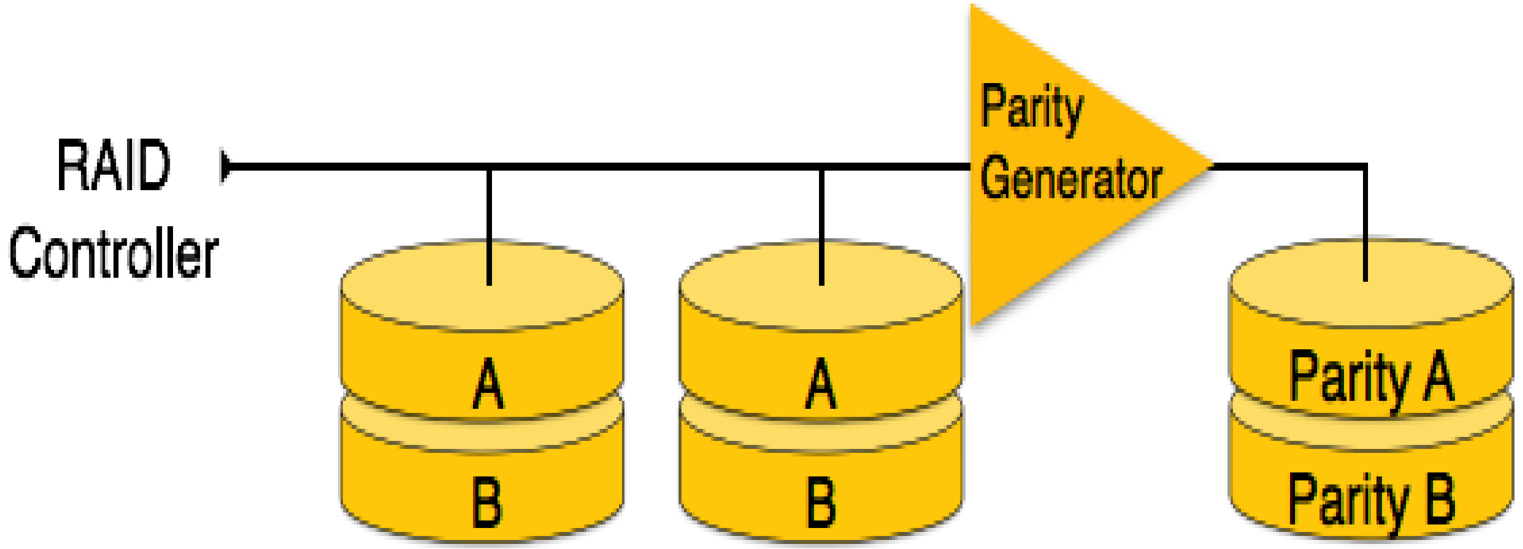
RAID 2

- RAID 2 records Error Correction Code using Hamming distance for its data, striped on different disks.
- Like level 0, each data bit in a word is recorded on a separate disk and ECC codes of the data words are stored on a different set disks.
- Due to its complex structure and high cost, RAID 2 is not commercially available.



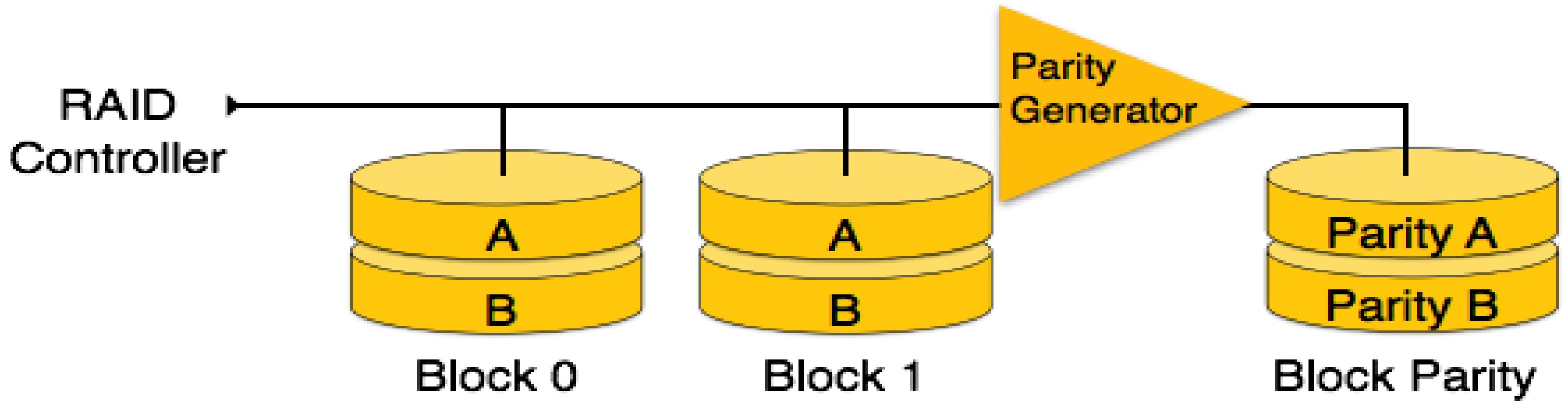
RAID 3

- RAID 3 stripes the data onto multiple disks.
- The parity bit generated for data word is stored on a different disk. This technique makes it to overcome single disk failures.



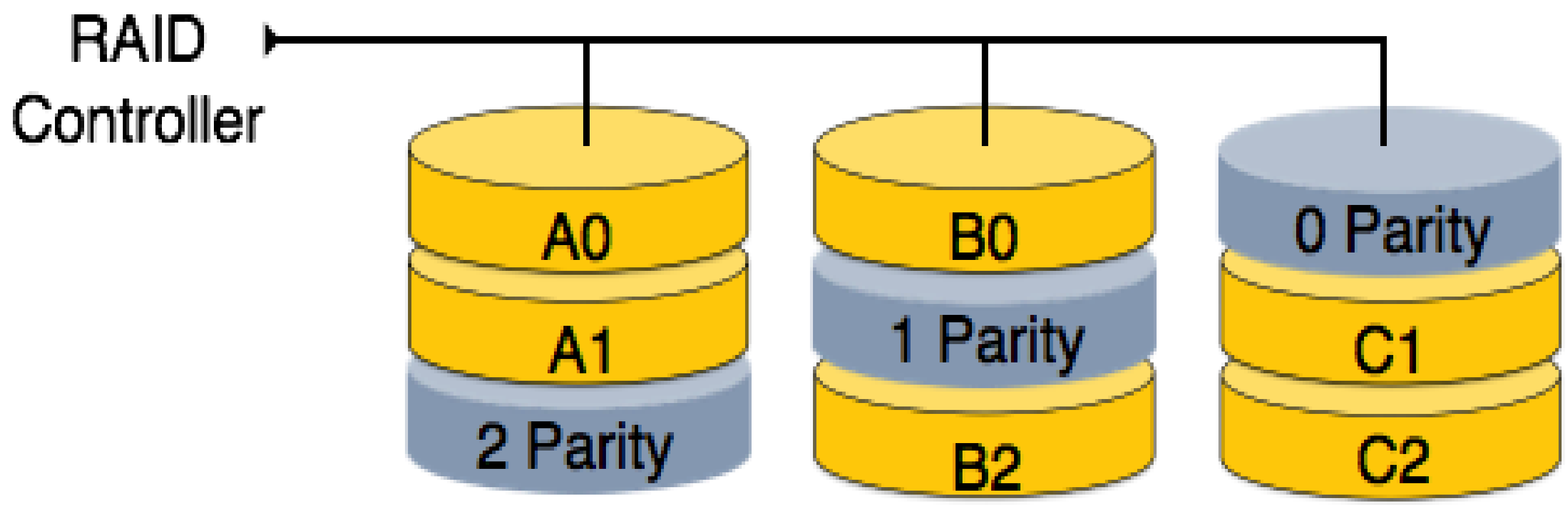
RAID 4

- In this level, an entire block of data is written onto data disks and then the parity is generated and stored on a different disk.
- Note that level 3 uses byte-level striping, whereas level 4 uses block-level striping.
- Both level 3 and level 4 require at least three disks to implement RAID.



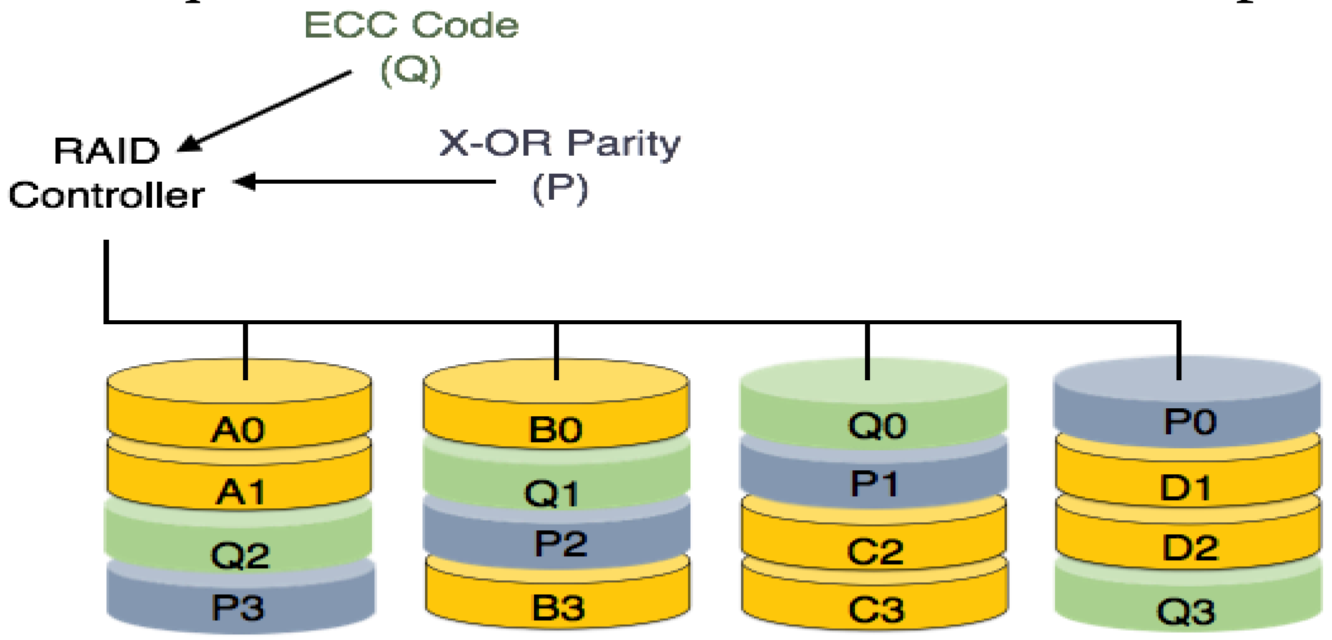
RAID 5

- RAID 5 writes whole data blocks onto different disks, but the parity bits generated for data block stripe are distributed among all the data disks rather than storing them on a different dedicated disk.



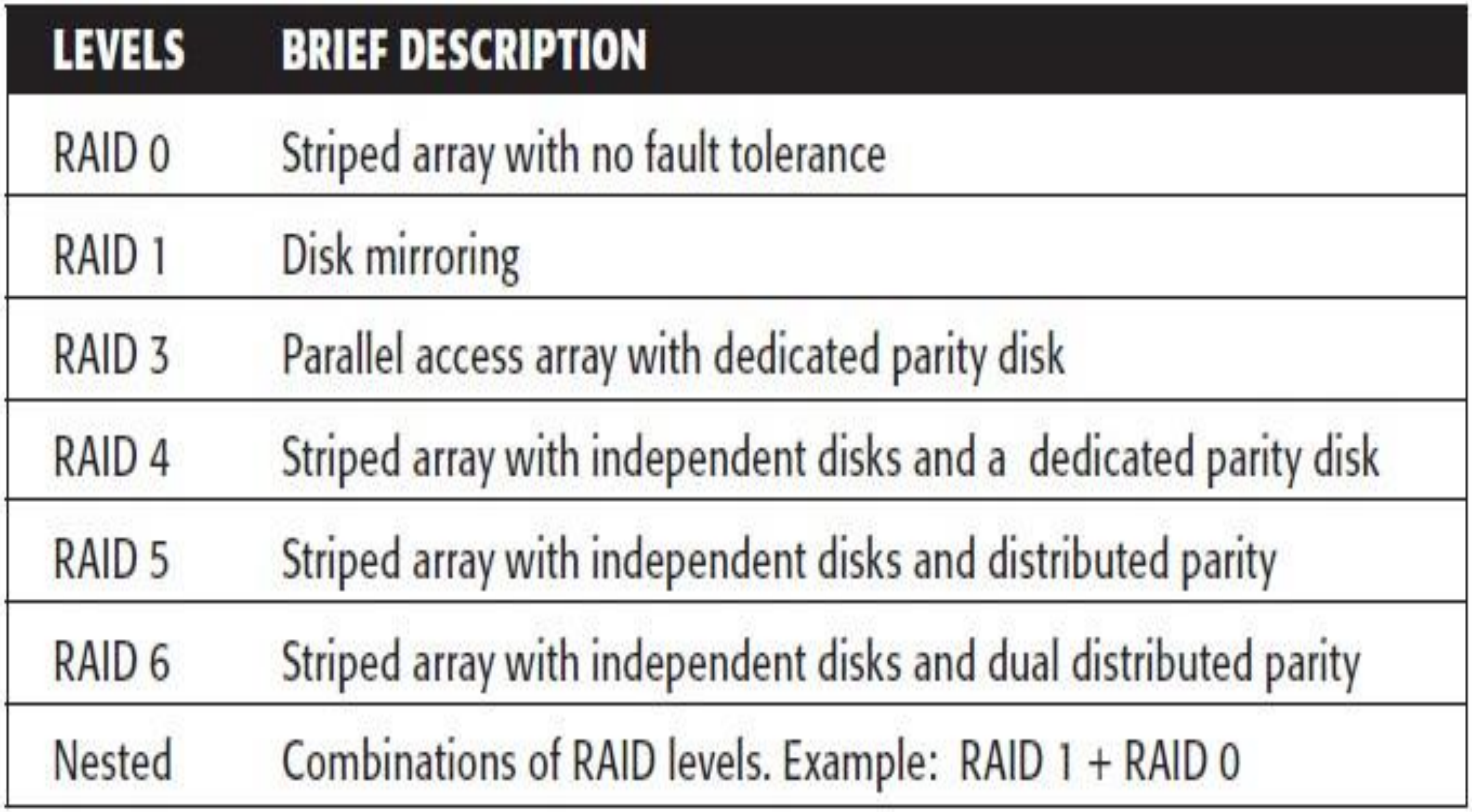
RAID 6

- RAID 6 is an extension of level 5.
- In this level, two independent parities are generated and stored in distributed fashion among multiple disks.
- Two parities provide additional fault tolerance.
- This level requires at least four disk drives to implement RAID.



RAID Levels

Table 3-1: Raid Levels



LEVELS	BRIEF DESCRIPTION
RAID 0	Striped array with no fault tolerance
RAID 1	Disk mirroring
RAID 3	Parallel access array with dedicated parity disk
RAID 4	Striped array with independent disks and a dedicated parity disk
RAID 5	Striped array with independent disks and distributed parity
RAID 6	Striped array with independent disks and dual distributed parity
Nested	Combinations of RAID levels. Example: RAID 1 + RAID 0

Striping

- A RAID set is a group of disks. Within each disk,
- a predefined number of contiguously addressable disk blocks are defined as strips.
- The set of aligned strips that spans across all the disks within the RAID set is called a stripe.

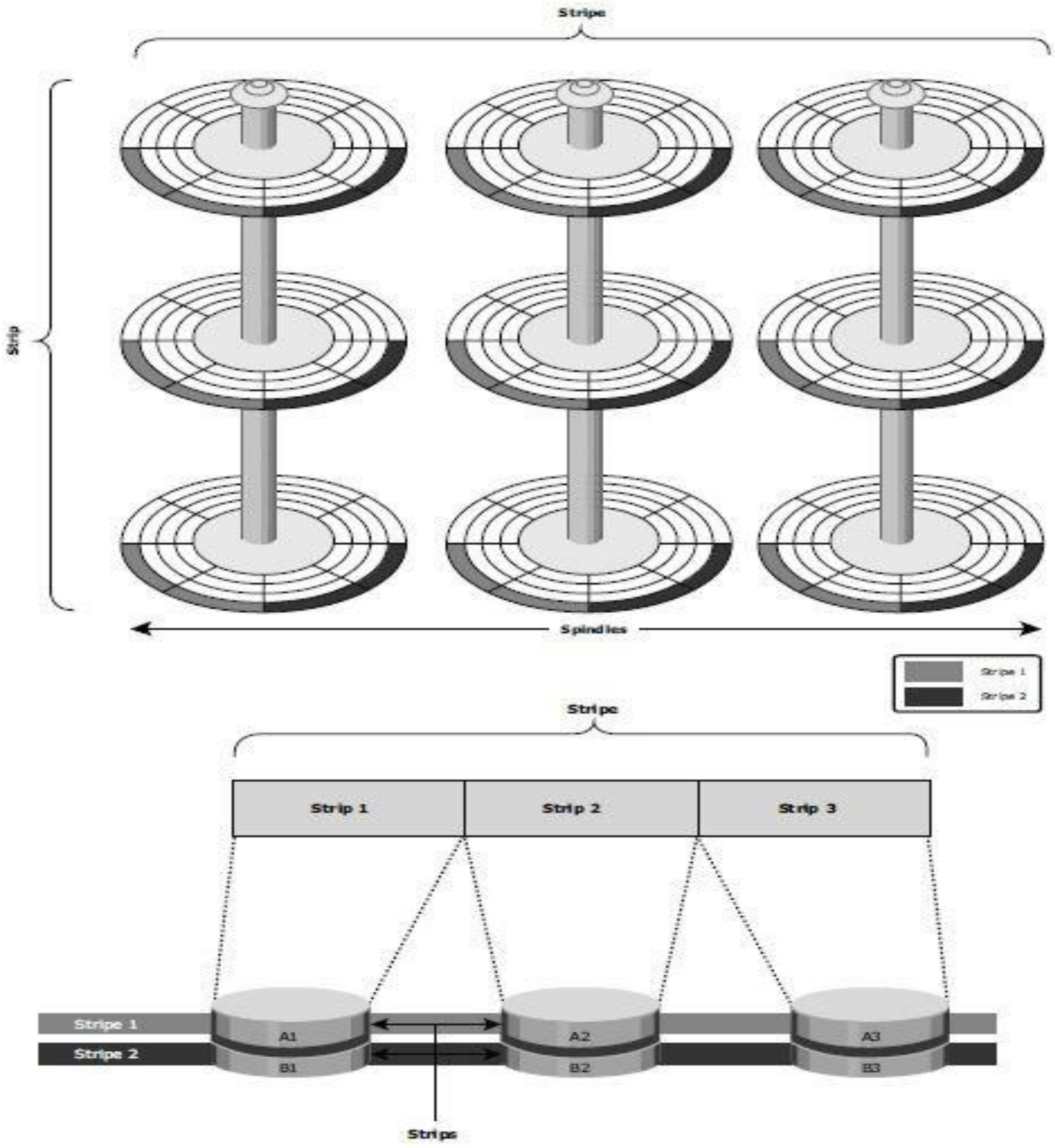


Figure 3-2: Striped RAID set

Mirroring

- data is stored on two different HDDs, yielding two copies of data.
- In the event of one HDD failure, the data is intact on the surviving HDD

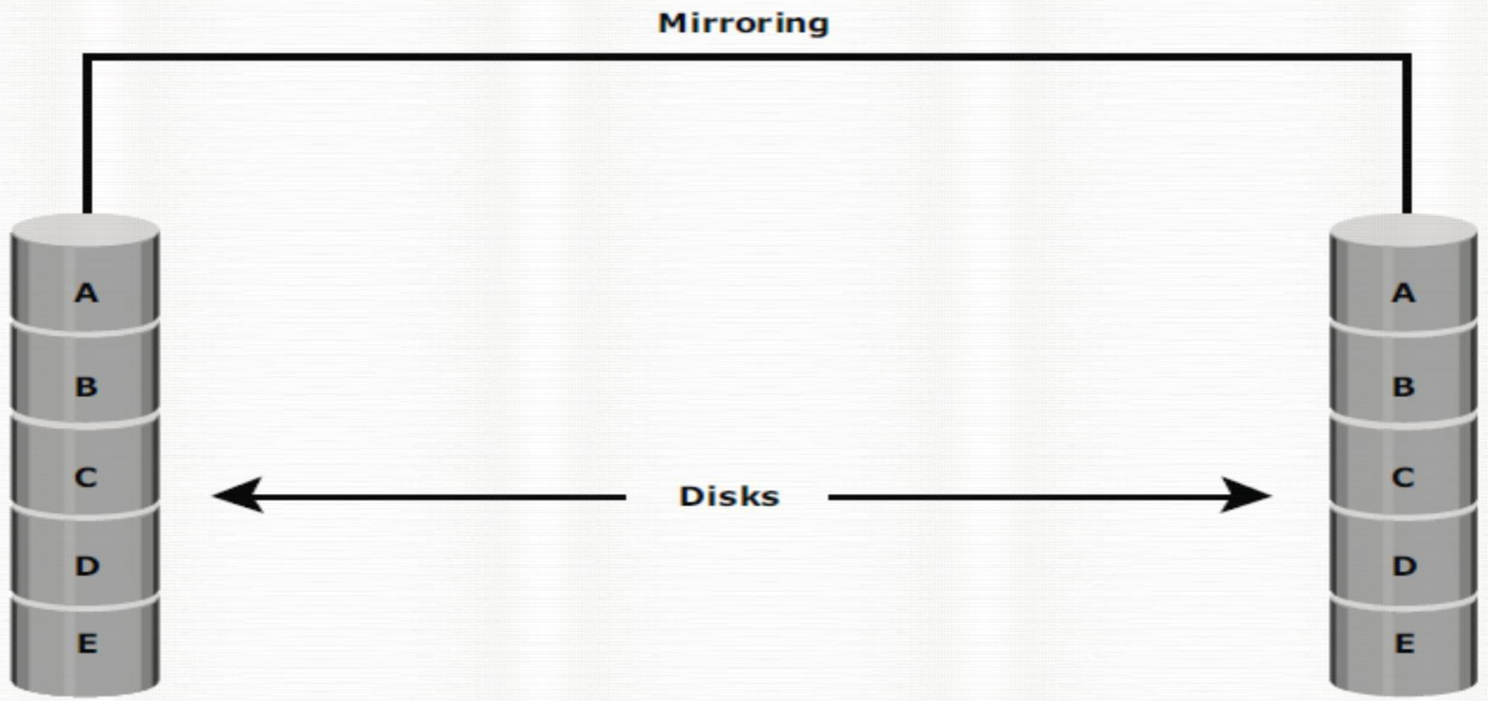


Figure 3-3: Mirrored disks in an array

Parity

- Parity is a method of protecting striped data from HDD failure without the cost of mirroring.
- An additional HDD is added to the stripe width to hold parity,
- a mathematical construct that allows re-creation of the missing data.
- Parity is a redundancy check that ensures full protection of data without maintaining a full set of duplicate data.

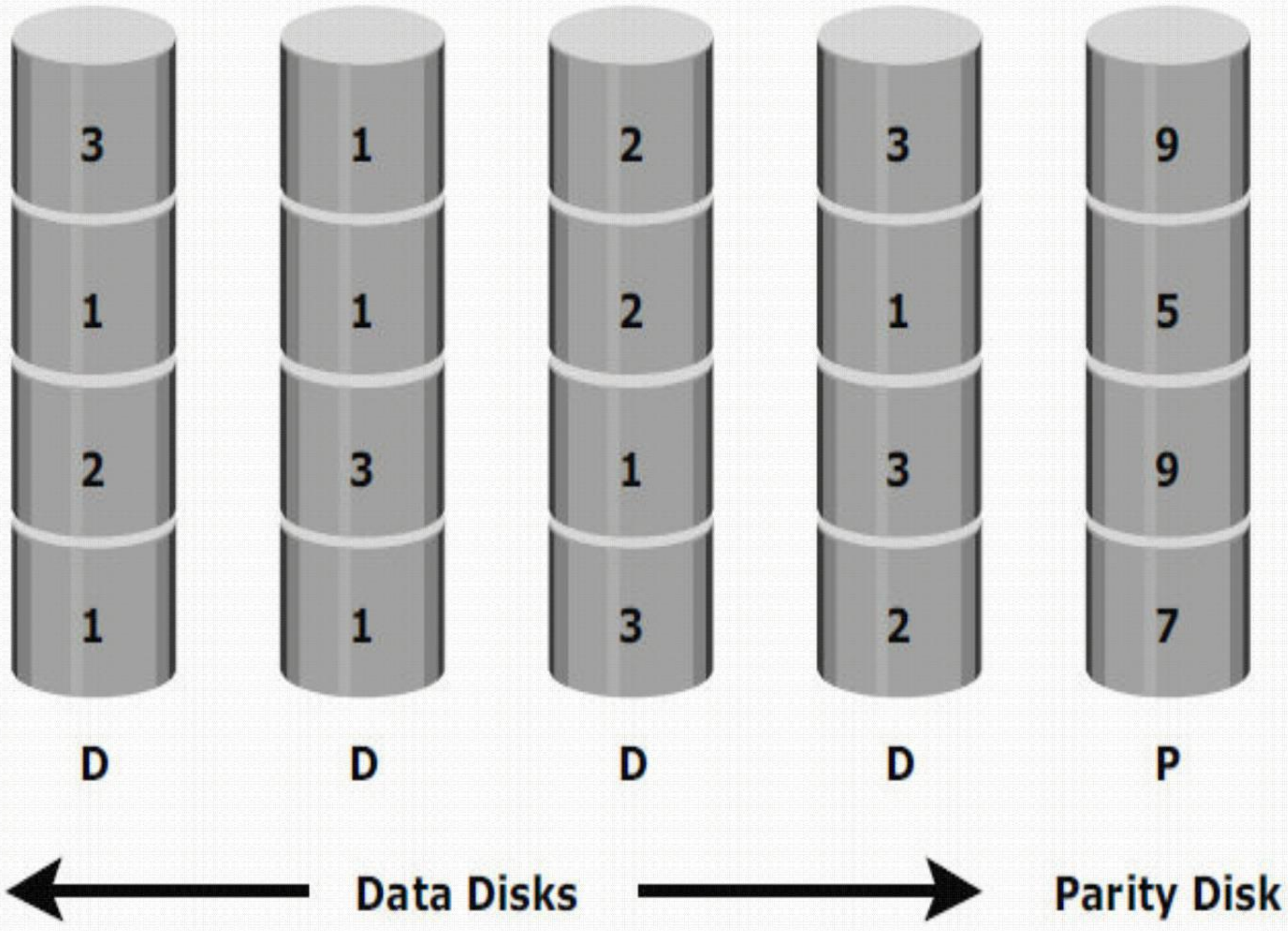


Figure 3-4: Parity RAID

Storage Access

- A database is mapped into a number of different files, which are maintained by the underlying operating system.
- These files reside permanently on disks, with backups on tapes.
- Each file is partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer

File Organization

- A **file** is organized logically as a sequence of records.
- These records are mapped onto disk blocks.
- Files are provided as a basic construct in operating systems

- blocks are of a fixed size determined by the physical properties of the disk and by the operating system,
- record sizes vary
- In a relational database, tuples of distinct relations are generally of different sizes.

- One approach to mapping the database to files is to use several files,
- and to store records of only one fixed length in any given file.
- An alternative is to structure our files so that we can accommodate multiple lengths for records;
- files of fixed length records are easier to implement than are files of variable-length records

Organization of records into blocks

- Data Items
- Records
- Files
- Blocks
- Memory

Fixed-Length Records

- type deposit = record

  account-number : char(10);

  branch-name : char (22);

  balance : real;

End

- Total size of one record is 40 bytes

- Problems
- **1.** It is difficult to delete a record from this structure.
- The space occupied by the record to be deleted must be filled with some other record of the file
- **2.** Unless the block size happens to be a multiple of 40 (which is unlikely),
- Some records will cross block boundaries.
- part of the record will be stored in one block and part in another.

Variable-Length Records

- Variable-length records arise in database systems in several ways:
- Storage of multiple record types in a file
- Record types that allow variable lengths for one or more fields
- Record types that allow repeating fields

- type account-list = record
- branch-name : char (22);
- account-info : array [1 ..∞] of
- record;
- account-number : char(10);
- balance : real;
- End
- End

- define account-info as an array with an arbitrary number of elements.
- the type definition does not limit the number of elements in the array,
- There is no limit on how large a record can be

Organization of Records in Files

- possible ways of organizing records in files are

- **Heap file organization.**
- Any record can be placed anywhere in the file where there is space for the record.
- ordering of records.

- **Sequential file organization.**
- Records are stored in sequential order, according to the value of a “search key” of each record.

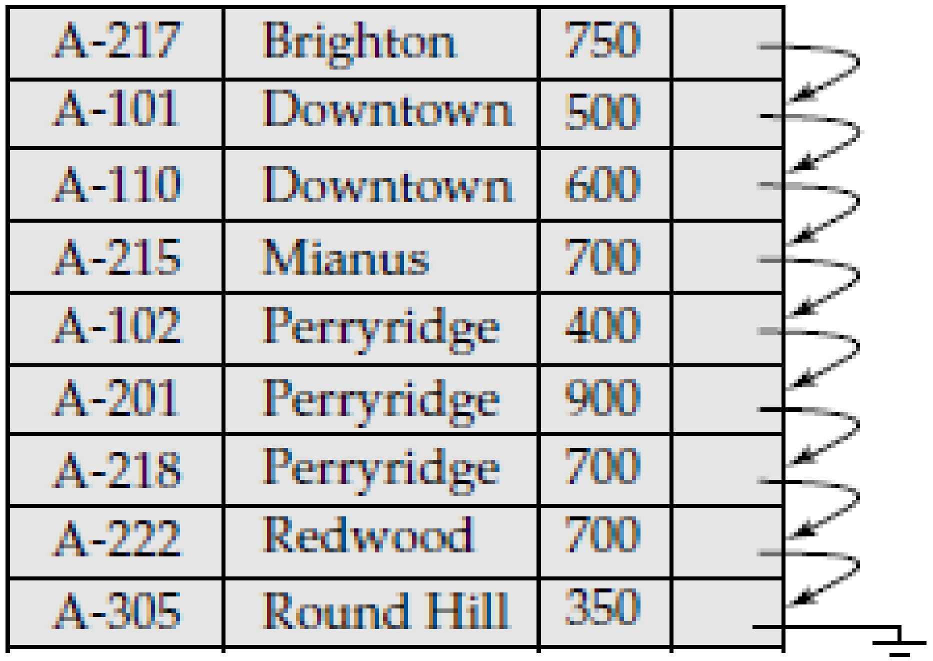
- **Hashing file organization**.
- A hash function is computed on some attribute of each record.
- The result of the hash function specifies in which block of the file the record should be placed.

Sequential File Organization

- A **sequential file** is designed for efficient processing of records in sorted order based on some search-key.
- A **search key** is any attribute or set of attributes;

- Figure shows a sequential file of account records taken from our banking example.
- In that example, the records are stored in search-key order, using branch name as the search key.

- It is difficult, however, to maintain physical sequential order as records are inserted and deleted,
- since it is costly to move many record
- How to perform insert and delete operations?



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

Figure 11.15 Sequential file for *account* records.

Data-Dictionary Storage

- A relational-database system needs to maintain data about the relations, such as the schema of the relations.
- This information is called the **data dictionary**, or **system catalog**.
- Among the types of information that the system must store are these:

- Names of the relations
- Names of the attributes of each relation
- Domains and lengths of attributes
- Names of views defined on the database, and definitions of those views
- Integrity constraints (for example, key constraints)
- In addition, many systems keep the following data on

users of the system:

- Names of authorized users
- Accounting information about users
- Passwords or other information used to authenticate users

- Some database systems store this information by using special-purpose data structures and code.
- It is generally preferable to store the data about the database in the database itself.

- The exact choice of how to represent system data by relations must be made by the system designers.

- One possible representation, with primary keys underlined, is
- Relation-metadata (relation-name, number-of-attributes, storage-organization, location)
- Attribute-metadata (attribute-name, relation-name, domain-type, position, length)
- User-metadata (user-name, encrypted-password, group)
- Index-metadata (index-name, relation-name, index-type, index-attributes)
- View-metadata (view-name, definition)

Buffer Management

- A major goal of the database system is to minimize the number of block transfers between the disk and memory.
- One way to reduce the number of disk accesses is to keep as many blocks as possible in main memory.
- The goal is to maximize the chance that, when a block is accessed, it is already in main memory, and, thus, no disk access is required.

- Programs in a database system make on the buffer manager when they need a block from disk.
- If the block is already in the buffer,
- the buffer manager passes the address of the block in main memory to the requester.

- If the block is not in the buffer,
- the buffer manager first allocates space in the buffer for the block, throwing out some other block, if necessary, to make space for the new block.
- The thrown-out block is written back to disk only if it has been modified since the most recent time that it was written to the disk.
- Then, the buffer manager reads in the requested block from the disk to the buffer, and passes the address of the block in main memory to the requester.

- **Buffer replacement strategy**
- When there is no room left in the buffer,
- a block must be removed from the buffer before a new one can be read in.

Buffer-Replacement Policies

- Most operating systems use a **least recently used (LRU)** scheme,
- in which the block that was referenced least recently is written back to disk and is removed from the buffer.

- **Most Recently used (MRU)**

Indexing and Hashing

Prof. Vivek V. Kheradkar

Basic Concepts

- “Find the balance of account number 101”
- Search sequentially for account number in table
- Linear search
- Average case – (n/2) comparisons
- To allow fast access, design additional structures associate with files.

- Index –
- E.g. Index in textbook

- Words in the index are in sorted order,making it easy to find the word we are looking for.
- index is much smaller than the book,
- further reducing the effort needed to find the words we are looking for.
- Card catalogs in libraries work in a similar manner

- Database system indices play the same role as book indices or card catalogs in libraries.
- For example, to retrieve an account record given the account number,
- The database system would look up an indexto find on which disk block the corresponding record resides, and then fetch the disk block, to get the account record.

- Keeping a sorted list of account numbers would not work well on very large databases with millions of accounts,
- since the index would itself be very big even though keeping the index sorted reduces the search time,
- finding an account can still be time consuming.
- more sophisticated indexing techniques may be used

- There are two basic kinds of indices:
- Ordered indices:
- Based on a sorted ordering of the values.

- Hash indices:
- Based on a uniform distribution of values across a range of buckets.
- The bucket to which a value is assigned is determined by a function, called a hash function

- Consider several techniques for both ordered indexing and hashing.
- No one technique is the best.
- each technique is best suited to particular database applications.
- Each technique must be evaluated on the basis of these factors:

- Access types:
- The types of access that are supported efficiently.
- Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.

- Access time:
- The time it takes to find a particular data item, or set of items

- Insertion time:
- Time it takes to insert a new data item.
- This value includes the time it takes to find the correct place to insert the new data item, and the time it takes to update the index structure.

- Deletion time:
- The time it takes to delete a data item.
- This value includes the time it takes to find the

item to be deleted, and The time it takes to update the index structure

- Space overhead:
- The additional space occupied by an index structure.
- Provided that the amount of additional space is moderate,
- it is usually worth-while to sacrifice the space to achieve improved performance.

- We often want to have more than one index for a file.
- For example, libraries maintained several card catalogs: for author, for subject, and for title

- An attribute or set of attributes used to look up records in a file is called a search key.

Ordered Indices

- To gain fast random access to records in a file,
- we can use an index structure.
- Each index structure is associated with a particular search key.
- an ordered index stores the values of the search keys in sorted order,
- and associates with each search key the records that contain it.

- The records in the indexed file may themselves be stored in some sorted order
- A file may have several indices, on different search keys.
- If the file containing the records is sequentially ordered, a primary index is an index whose search key also defines the sequential order of the file.

- Primary indices are also called clustering indices.
- The search key of a primary index is usually the primary key, although that is not necessarily so.
- Indices whose search key specifies an order different from the sequential order of the
- file are called secondary indices, or non-clustering indices.

Primary Index

- Assume that all files are ordered sequentially on some search key.
- Such files, with a primary index on the search key, are called index-sequential files.
- They represent one of the oldest index schemes used in database systems.
- They are designed for applications that require both sequential processing of the entire file and
- random access to individual records.
- Primary index is also called as clustering index

Dense and Sparse Indices

- An index record, or index entry, consists of a search-key value, and
- pointers to one or more records with that value as their search-key value.
- The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

- There are two types of ordered indices:

- Dense index:
- An index record appears for every search-key value in the file.
- In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search- key value.
- The rest of the records with the same search key-value would be stored sequentially after the first record,

- because the index is a primary one, records are sorted on the same search key
- Dense index implementations may store a list of pointers to all records with the same search-key value;
- doing so is not essential for primary indices

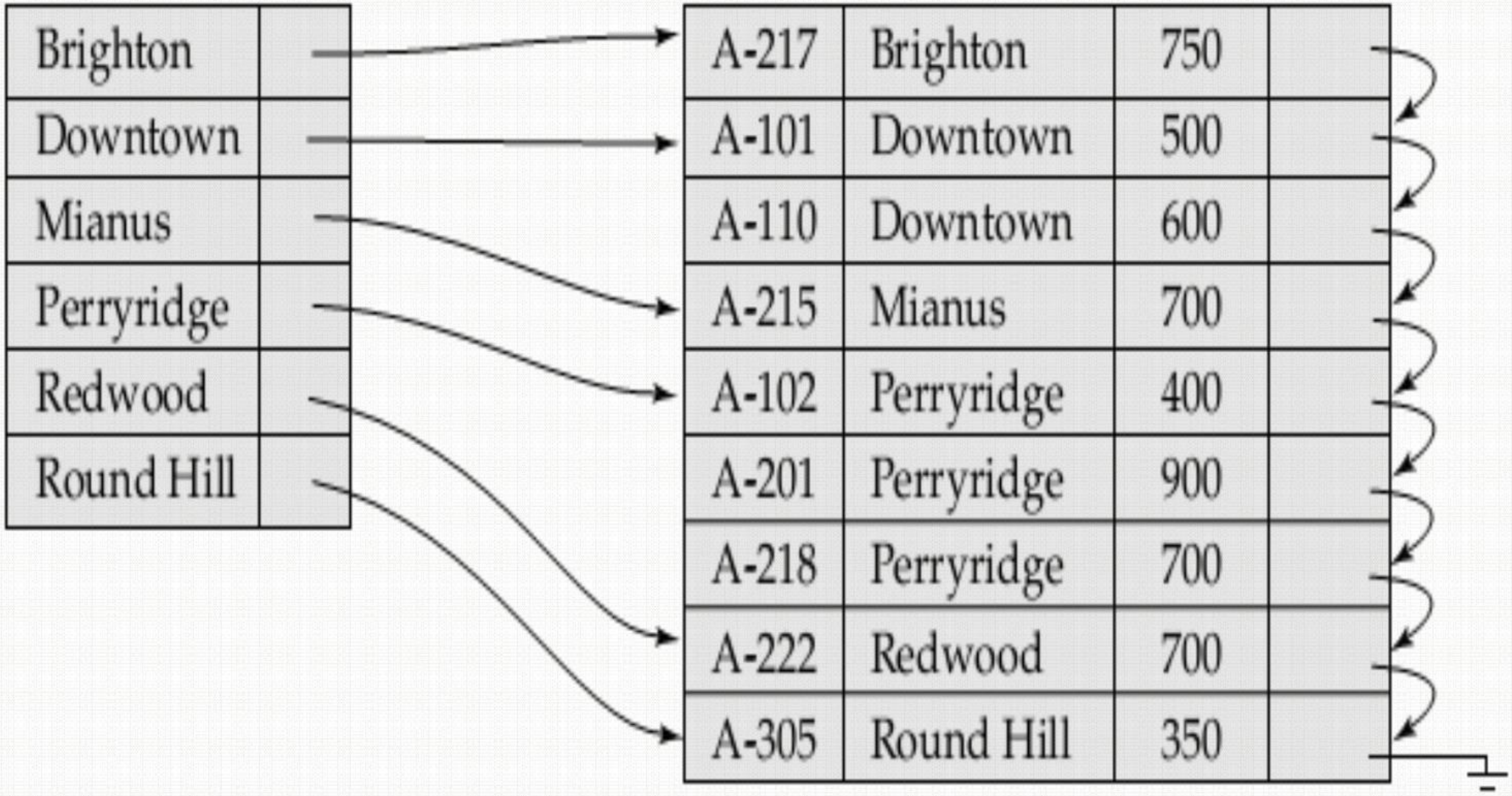


Figure 12.2 Dense index.

- Sparse index:
- An index record appears for only some of the search-key values.
- As is true in dense indices, each index record contains a search-key value and a pointer to the first data record with that search- key value.
- To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking.

- We start at the record pointed to by that index entry,
- and follow the pointers in the file until we find the desired record.

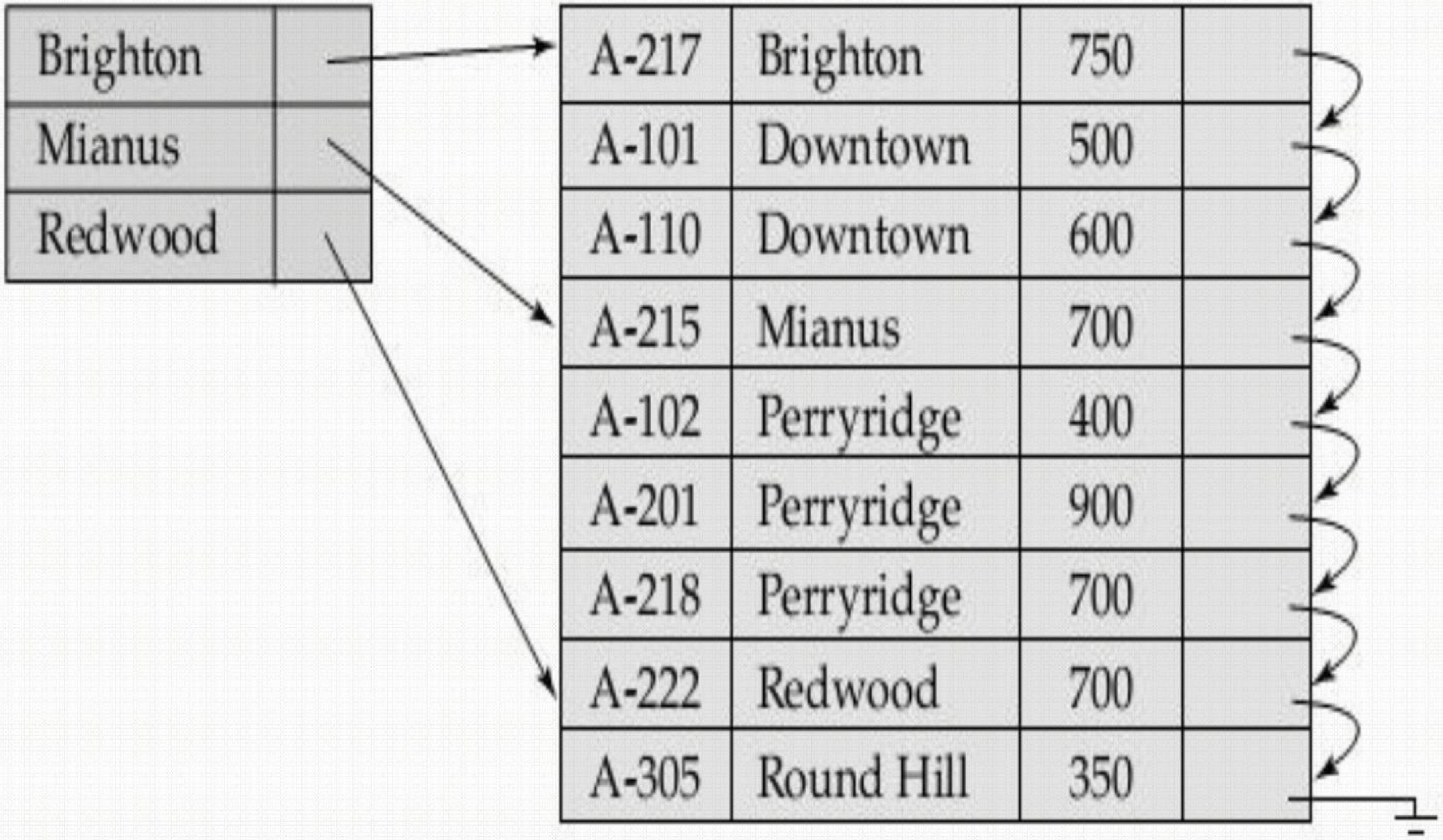


Figure 12.3 Sparse index.

Dense and Sparse Indices Comparison

- it is generally faster to locate a record if we have a dense index rather than a sparse index.
- However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

- There is a trade-off that the system designer must make between access time and space overhead.
- Although the decision regarding this trade-off depends on the specific application,
- a good compromise is to have a sparse index with one index entry per block.
- Once we have brought the block in main memory, the time to scan the entire block is negligible.
- Using this sparse index, we locate the block containing the record that we are seeking.

Multilevel Indices

- If we use a sparse index, the index itself may become too large for efficient processing.
- If an index is sufficiently small to be kept in main memory, the search time to find an entry is low.
- If index is so large that it must be kept on disk,
- a search for an entry requires several disk block reads.
- Binary search can be used on the index file to locate an entry, but the search still has a large cost.

- To deal with this problem,
- we treat the index just as we would treat any other sequential file,
- Construct a sparse index on the primary index
- To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire.
- The pointer points to a block of the inner index.
- We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire.

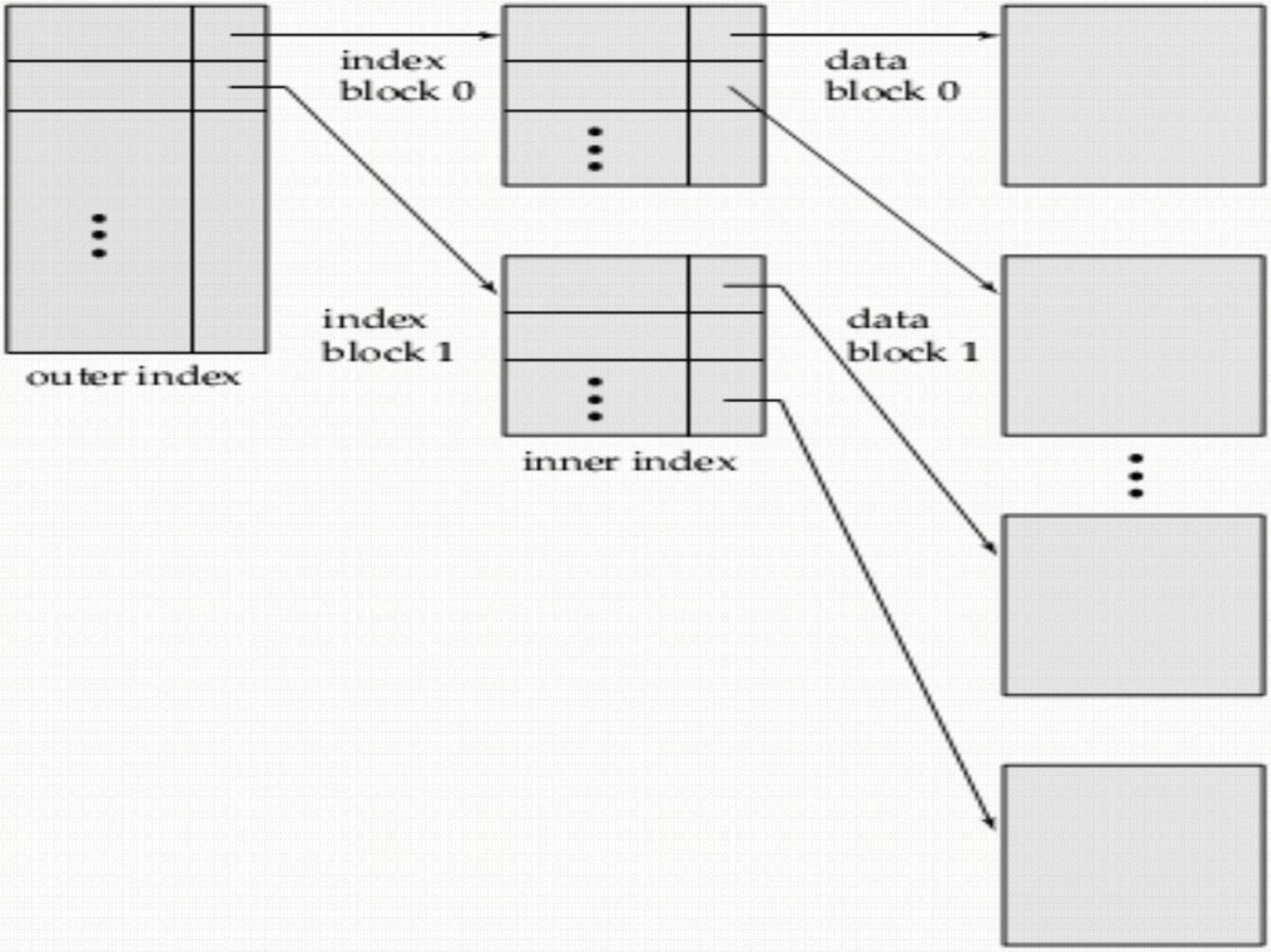


Figure 12.4 Two-level sparse index.

- The pointer in this record points to the block of the file that contains the record for which we are looking.
- we can repeat this process as many times as necessary.
- Indices with two or more levels are called multilevel indices.

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.
- Insertion- First, the system performs a lookup using the search-key value that appears in the record to be inserted.
- Deletion- To delete a record, the system first looks up the record to be deleted.
- actions the system takes next depend on whether the index is dense or sparse.

Insertion

- **Dense indices:**

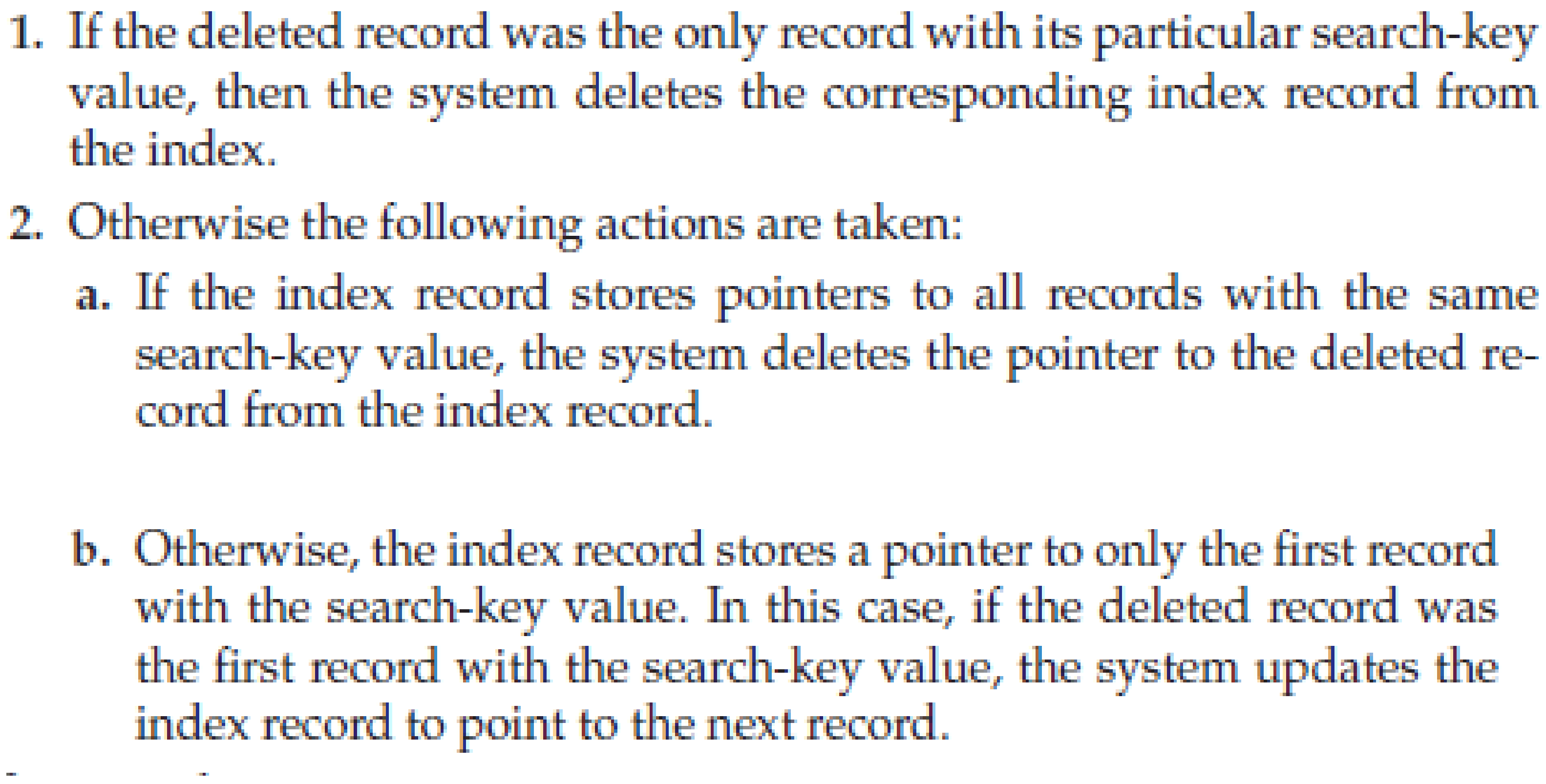
1. If the search-key value does not appear in the index, the system inserts an index record with the search-key value in the index at the appropriate position.
2. Otherwise the following actions are taken:
   a. If the index record stores pointers to all records with the same search-key value, the system adds a pointer to the new record to the index record.
   b. Otherwise, the index record stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.

- **Sparse indices:**

•We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value appearing in the new block into the index

Deletion

- **Dense indices:**



1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index record from the index.
2. Otherwise the following actions are taken:
   a. If the index record stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index record.

   b. Otherwise, the index record stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index record to point to the next record.

Deletion

- **Sparse indices:**

1. If the index does not contain an index record with the search-key value of the deleted record, nothing needs to be done to the index.
2. Otherwise the system takes the following actions:
   a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
   b. Otherwise, if the index record for the search-key value points to the record being deleted, the system updates the index record to point to the next record with the same search-key value.

Secondary Indices

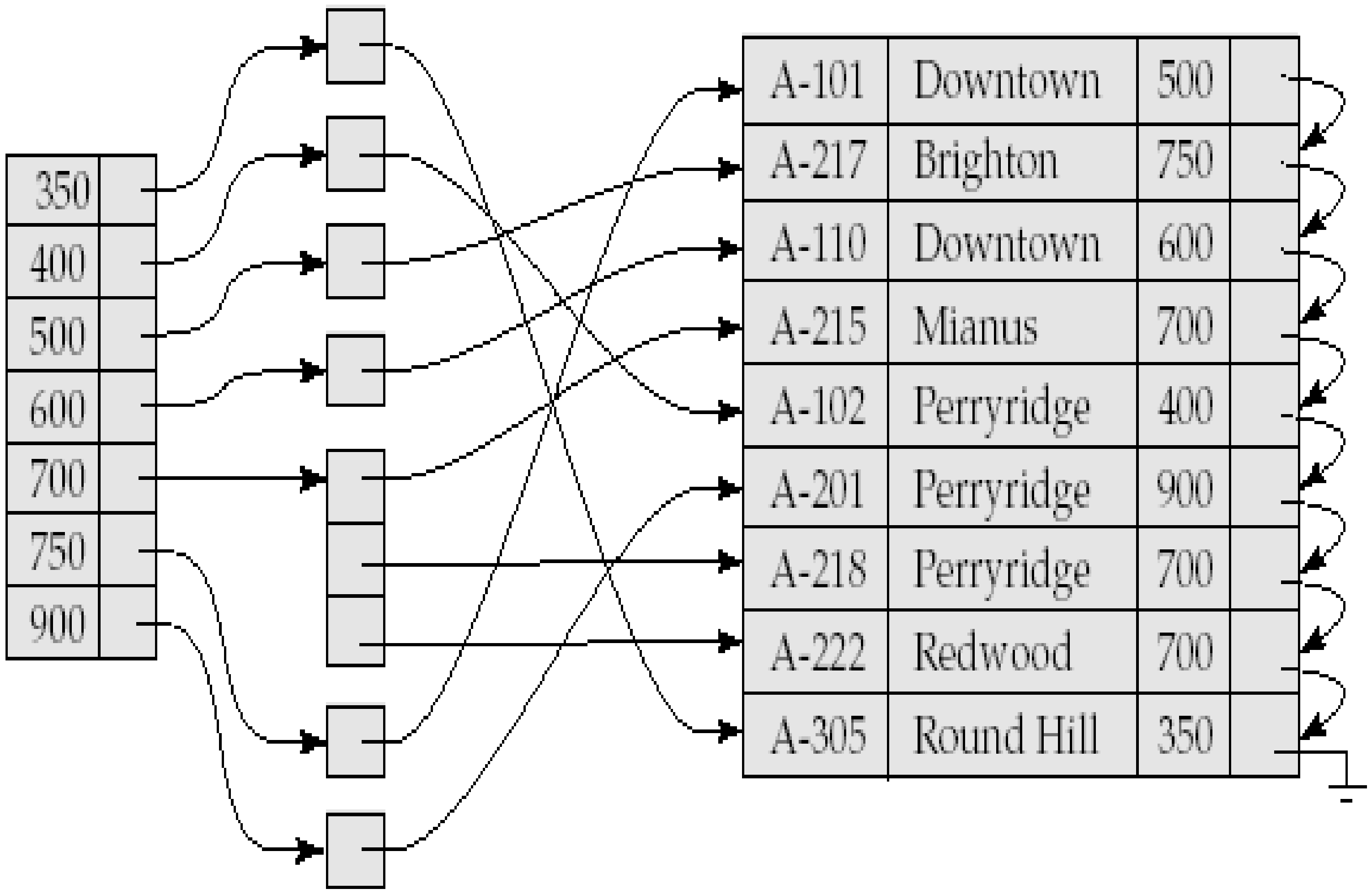
- Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file.
- A primary index may be sparse, storing only some of the search-key values,
- since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file.
- Secondary index is also called as non-clustering index

- If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and we cannot find them without searching the entire file.

- A secondary index on a candidate key looks just like a dense primary index, except that the records pointed to by successive values in the index are not stored sequentially.

- A secondary index must contain pointers to all the records.

- The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file.

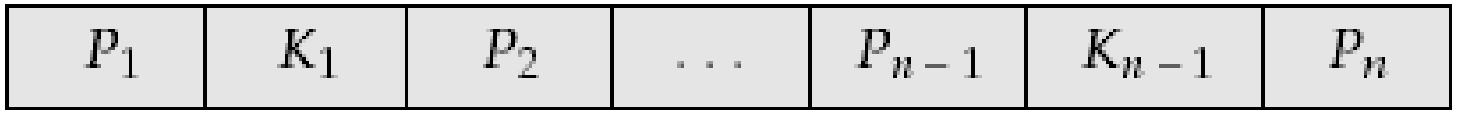


Secondary index on *account* file, on noncandidate key *balance*.

B+-Tree Index Files

- The main disadvantage of the index-sequential file organization is that performance degrades as the file grows,
- The B+-tree index structure is the most widely used of several index structures that maintain their efficiency
- B+-tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.

- Each nonleaf node in the tree has between [n/2] and n children,
- **Structure of a B+-Tree :**
- A B+-tree index is a multilevel index,

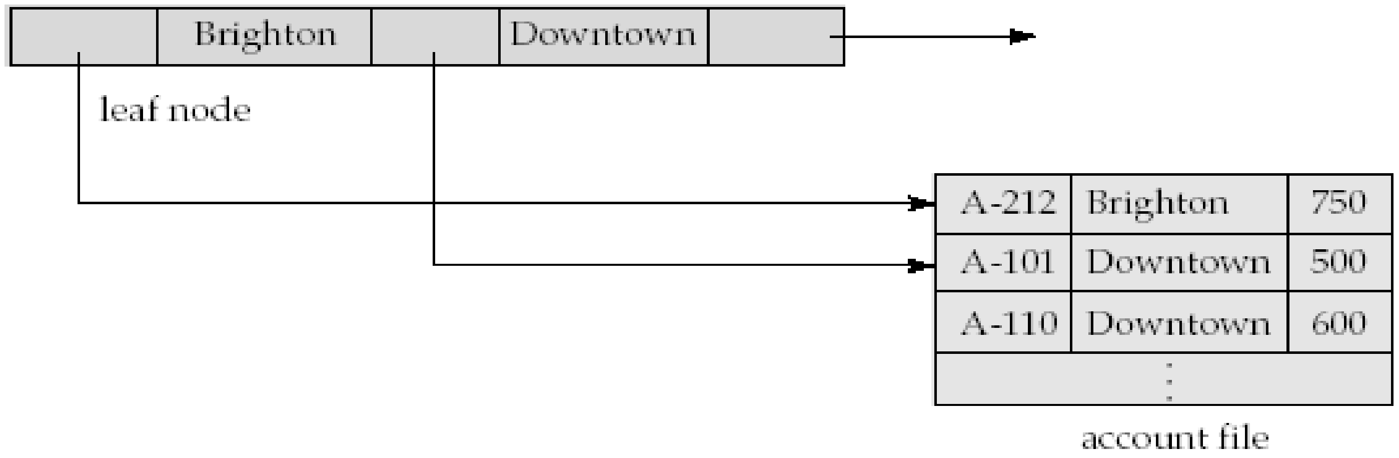


Typical node of a B^+-tree.

- It contains up to n - 1 search-key values,

  $K_1, K_2, \ldots, K_{n-1}$
- *n* pointers $P_1, P_2, \ldots, P_n$
- The search-key values within a node are kept in sorted order. if $i < j$, then $K_i < K_j$

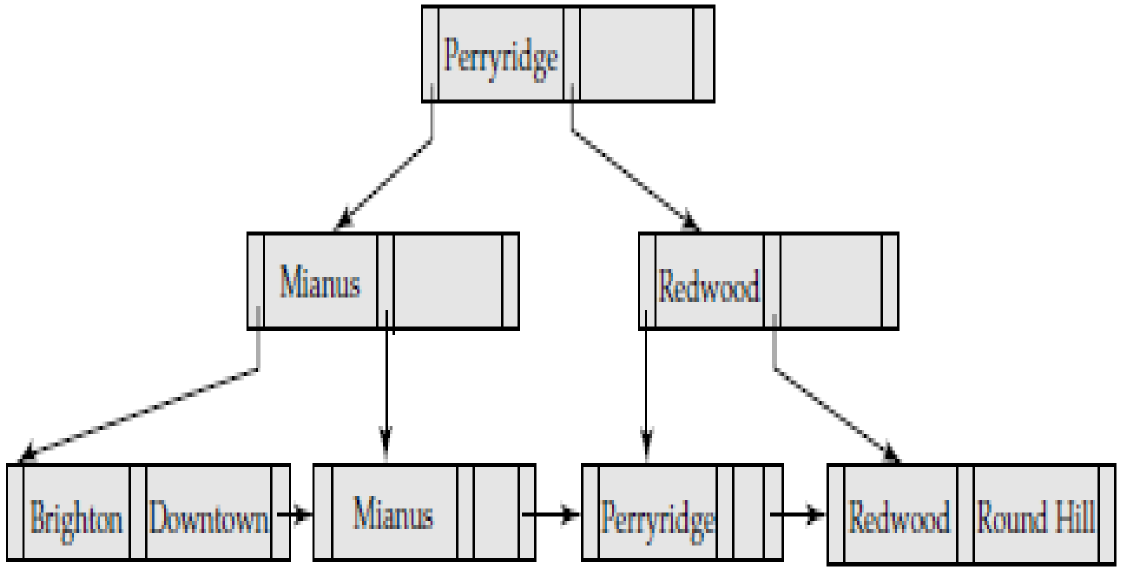
- Shows one leaf node of a B+-tree for the account file, in which we have chosen n to be 3, and the search key is branch-name.
- If Li & Lj -leaf node and i < j, then Li < Lj
- If the B+-tree index is to be a dense index, every search-key value must appear in some leaf node.



A leaf node for *account* B^+-tree index ($n = 3$).

- The nonleaf nodes of the B+-tree form a multilevel (sparse) index on the leaf nodes.
- The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes.
- A nonleaf node may hold up to n pointers, and must hold at least [n/2] pointers.
- The number of pointers in a node is called the *fanout* of the node.

- For leaf nodes, Pi (i = 1,..., 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. .



B$^+$-tree for *account* file ($n = 3$).

Search Queries on B+-Trees

- Suppose that we wish to find all records with a search-key value of V.
- First, we examine the root node, looking for the smallest search-key value greater than V .
- Suppose that we find that this search-key value is Ki .We then follow pointer Pi to another node
- If we find no such value, then k ≥ Km−1, where m is the number of pointers in the node

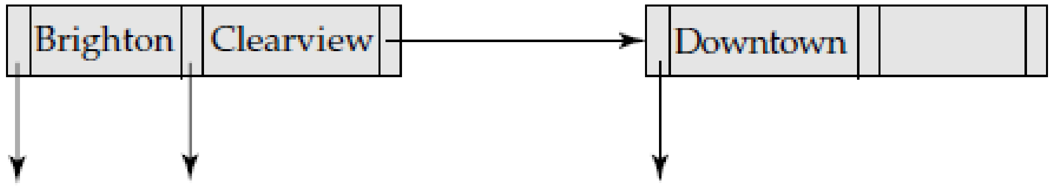
- In the node we reached above, again we look for the smallest search-key value greater than V, and once again follow the corresponding pointer as above.
- At the leaf node, if we find search key value Ki equals V , then pointer Pi directs us to the desired record or bucket.
- The value V is not found in the leaf node, no record with key value V exists.
- If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

Updates on B+-Trees

- **Insertion :**
- find the leaf node in which the search-key value would appear.
- If the search-key value already appears in the leaf node, we add the new record to the file and, if necessary, add to the bucket a pointer to the record
- If the search-key value does not appear, we insert the value in the leaf node, and position it such that the search keys are still in order

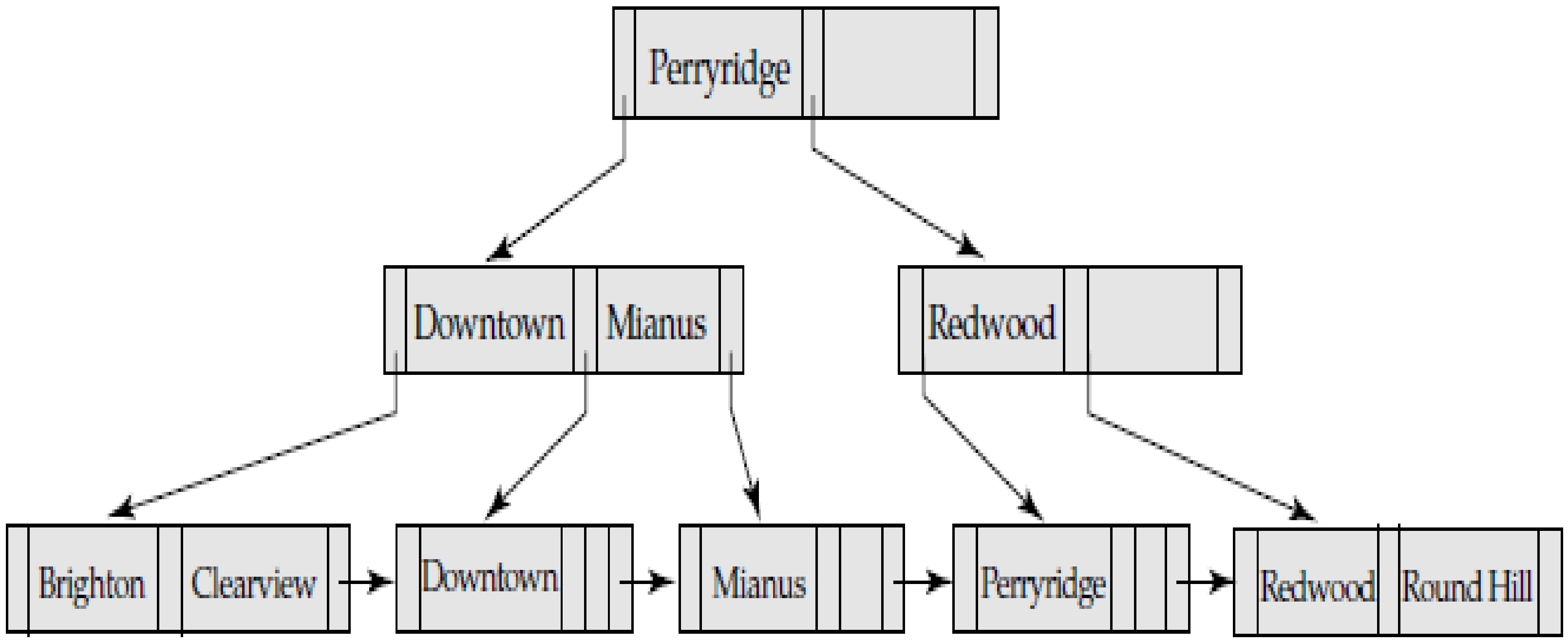
Updates on B+-Trees

- We then insert the new record in the file and, if necessary, create a new bucket with the appropriate pointer.

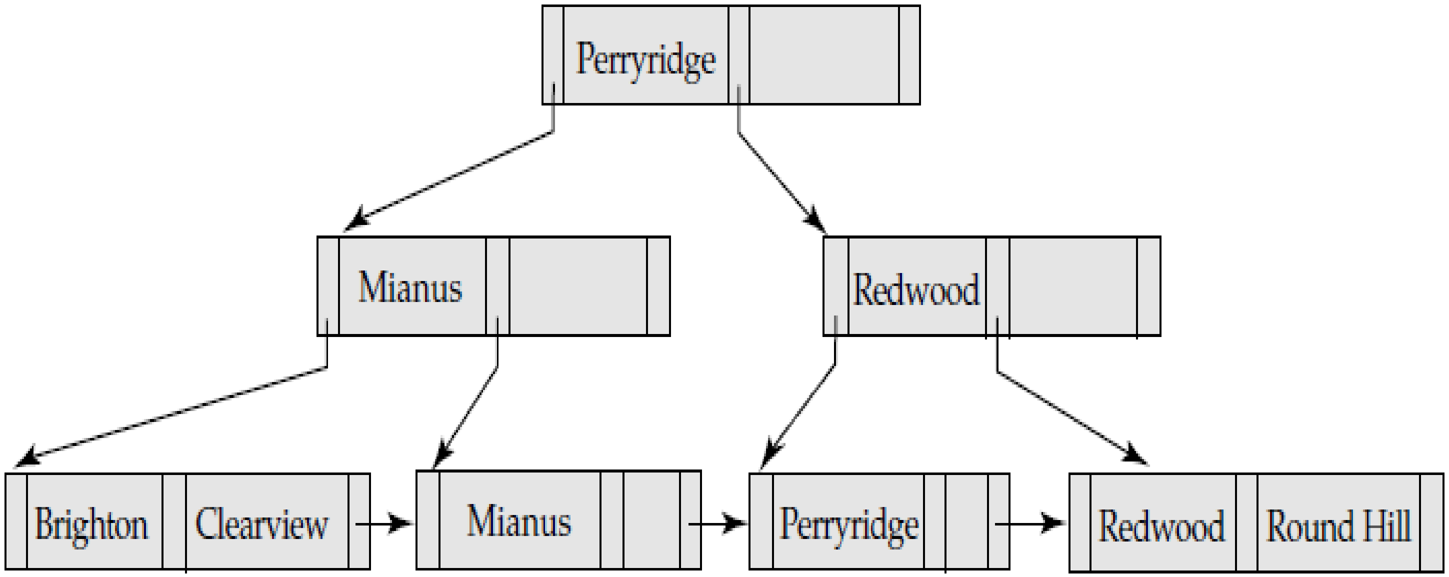


Split of leaf node on insertion of "Clearview."

- **<u>Deletion :</u>**
- find the record to be deleted, and remove it from the file..



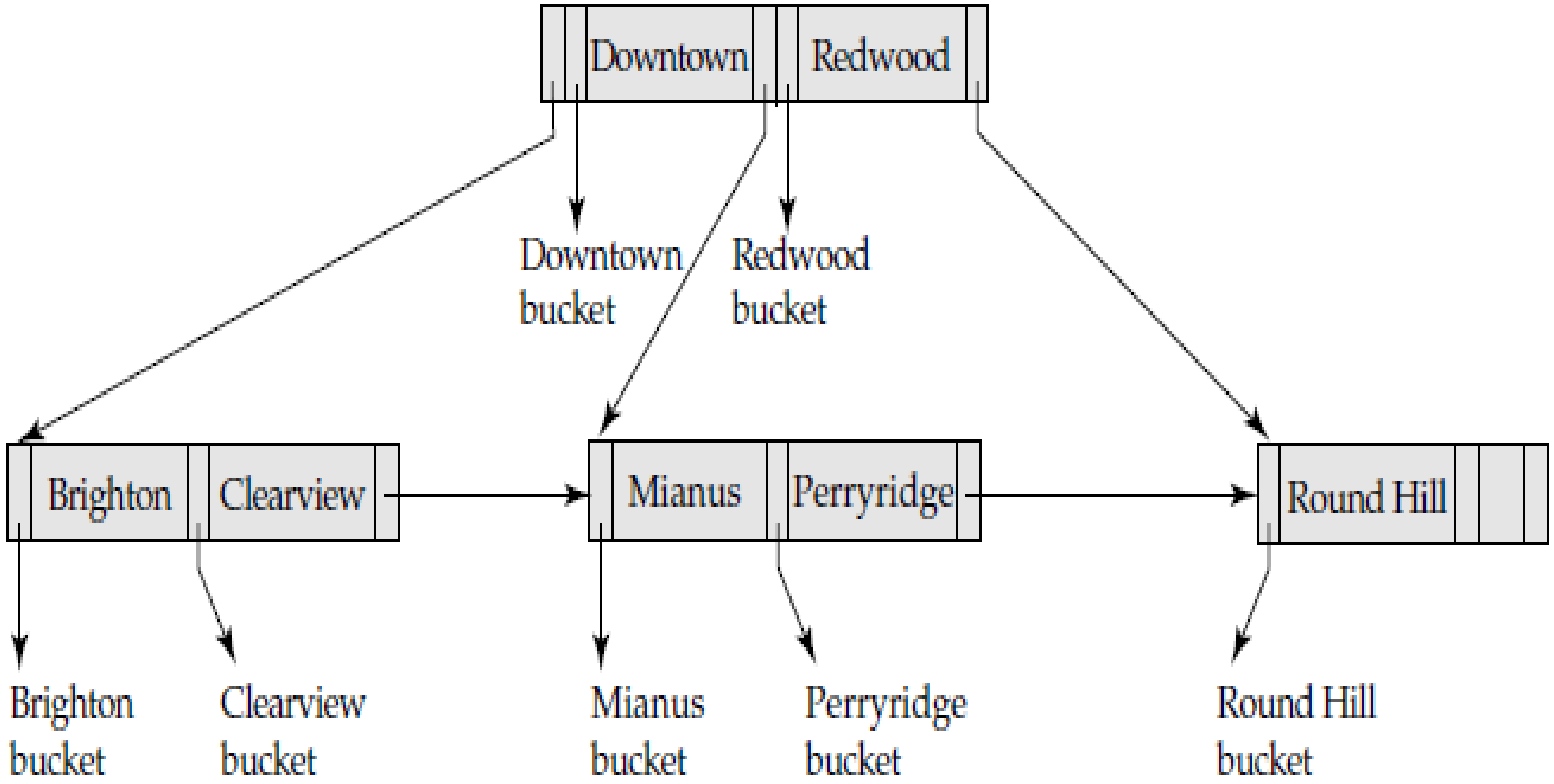
Insertion of “Clearview” into the B+-tree



Deletion of “Downtown” from the B$^+$-tree

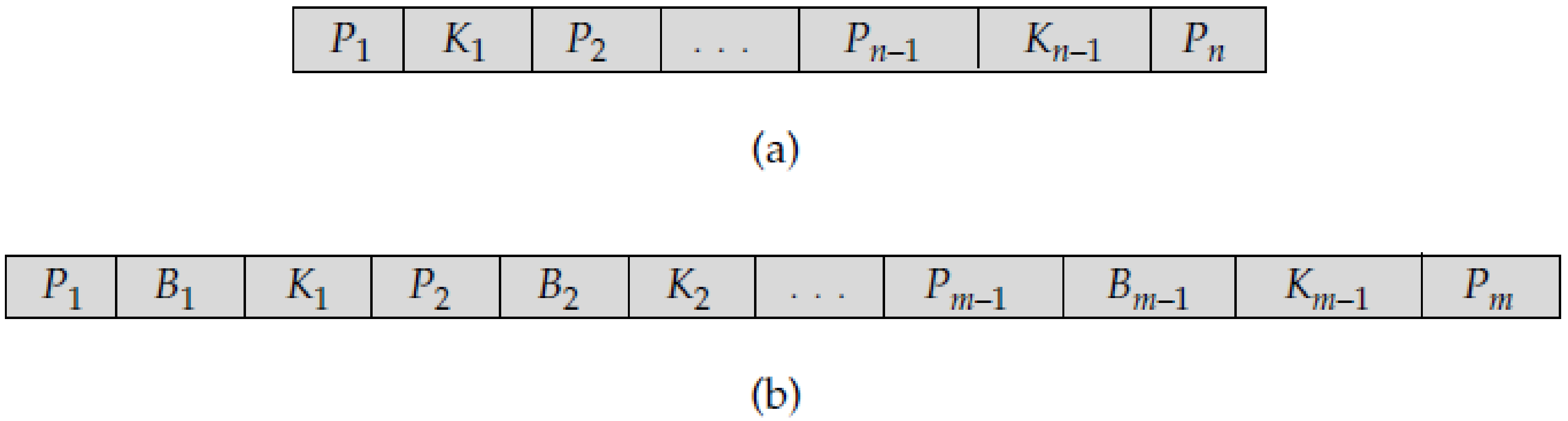
B-Tree Index Files

- The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values.
- A B-tree allows search-key values to appear only once



B-tree equivalent of B^+-tree

- keys that appear in nonleaf nodes appear nowhere else in the B-tree.
- an additional pointer field for each search key in a nonleaf node.
- additional pointers point to either file records or buckets for the associated search key.


P_1 | K_1 | P_2 | . . . | P_{n-1} | K_{n-1} | P_n
(a)
P_1 | B_1 | K_1 | P_2 | B_2 | K_2 | . . . | P_{m-1} | B_{m-1} | K_{m-1} | P_m
(b)

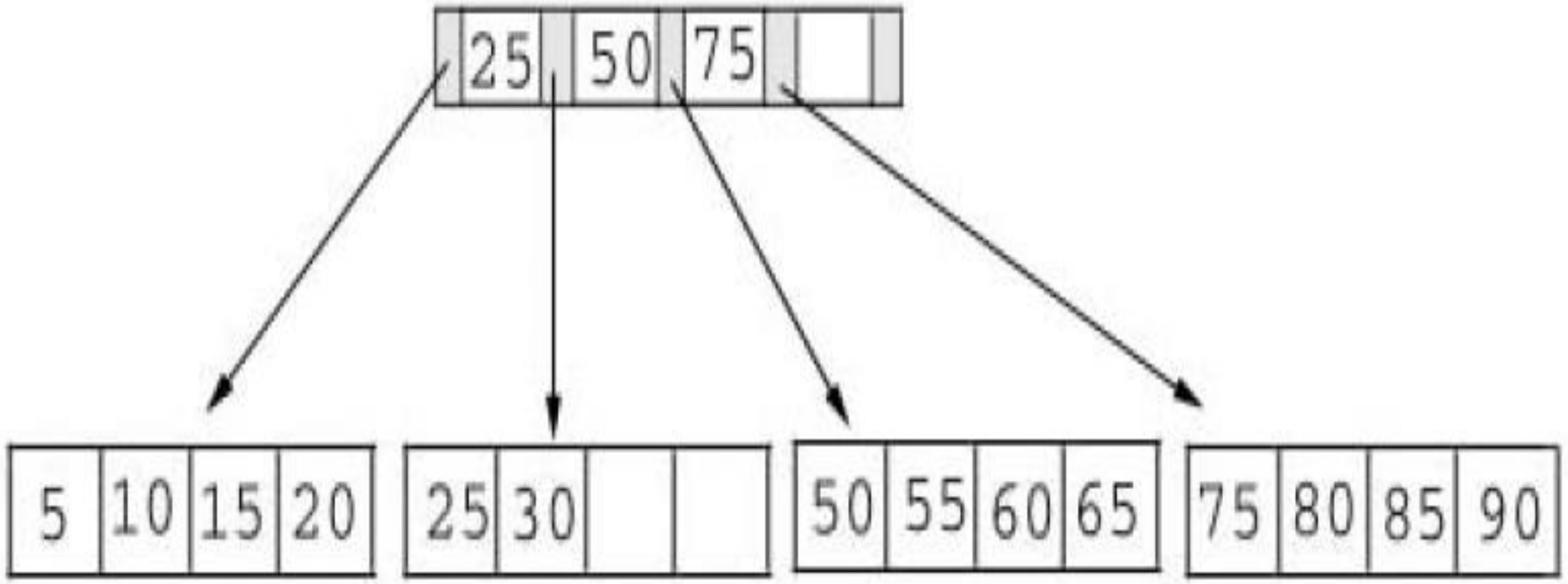
Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

- In non leaf node, node with two pointer Pi & Bi,
- Bi are bucket or file-record pointer
- The number of nodes accessed in a lookup in a B-tree depends on where the search key is located.
- A lookup on a B+-tree requires traversal of a path from the root of the tree to some leaf node
- A B-tree has a smaller fanout and therefore may have depth greater than that of the corresponding B+-tree.
- Deletion in a B-tree is more complicated.
- In a B+-tree, the deleted entry always appears in a leaf
- In a B-tree, the deleted entry may appear in a nonleaf node.

B+-Tree Example

- B+ Trees and B Trees use a "fill factor" to control the growth and the shrinkage.
- A 50% fill factor would be the minimum for any B+ or B tree. B+ tree conforms to the following guidelines
- Number of Keys/page 4
- Number of Pointers/page 5
- Fill Factor 50%
- Minimum Keys in each page 2

B+ Tree with four keys



The `insert` algorithm for B+ Trees

Leaf Page Full	**Index Page FULL**	**Action**
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	1. Split the leaf page 2. Place Middle Key in the index page in sorted order. 3. Left leaf page contains records with keys below the middle key. 4. Right leaf page contains records with keys equal to or greater than the middle key.
YES	YES	1. Split the leaf page. 2. Records with keys < middle key go to the left leaf page. 3. Records with keys >= middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

The `delete` algorithm for B+ Trees

Leaf Page Below Fill Factor	**Index Page Below Fill Factor**	**Action**
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling. Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.

Static Hashing

- One disadvantage of sequential file organization is that we must access an index structure
- to locate data, or must use binary search, and that results in more I/O operations.
- File organizations based on the technique of hashing allow us to avoid accessing an index structure.
- Hashing - one way that we could get `O(1)` access without wasting a lot of space?

Hash File Organization

- In a hash file organization,
- we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record.
- the term bucket denotes a unit of storage that can store one or more records.
- A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.

- Let K denote the set of all search-key values, and let B denote the set of all bucket addresses.
- A hash function h is a function from K to B.
- Let h denote a hash function.
- To insert a record with search key Ki,
- we compute h(Ki),
- which gives the address of the bucket for that record.
- Assume that there is space in the bucket to store the record.
- Then, the record is stored in that bucket.

- To perform a lookup on a search-key value Ki, we simply compute h(Ki),then
- search the bucket with that address.
- Suppose that two search keys,K5 and K7, have the same hash value;
- that is, h(K5)=h(K7).
- If we perform a look up on K5, the Bucket h(K5) contains records with search-key values K5 and records with search-key values K7.
- Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want

- Deletion is equally straightforward.
- If the search-key value of the record to be deleted is Ki,
- we compute h(Ki),
- then search the corresponding bucket for that record,
- and delete the record from the bucket.

Hash Functions

- Worst possible hash function maps all search-key values to the same bucket.
- Such a function is undesirable because all the records have to be kept in the same bucket.
- A lookup has to examine every such record to find the one desired.
- An ideal hash function distributes the stored keys uniformly across all the buckets,
- so that every bucket has the same number of records.

- Choose a hash function that assigns search-key values to buckets in such a way that the distribution has these qualities:

- The distribution is uniform.
- the hash function assigns 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 values assigned to it,

- regardless of the actual distribution of search-key values.

- Some hash functions
- Middle of square
- H(x):= return middle digits of x^2
- Division
- H(x):= return x % k
- Multiplicative:
- H(x):= return the first few digits of the fractional part of x*k, where k is a fraction.

Handling of Bucket Overflows

- If the bucket does not have enough space, a bucket over flow is said to occur.
- Bucket overflow can occur for several reasons:
- Insufficient buckets.
- Skew
- Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space.
- This situation is called bucket skew.

- Skew can occur for two reasons:
- 1. Multiple records may have the same search key.
- 2. The chosen hash function may result in non-uniform distribution of search keys.

- There are two types of hashing :

1. ***Static hashing***: In static hashing, the hash function maps search-key values to a fixed set of *locations*.

2. ***Dynamic hashing***: In dynamic hashing a hash table can grow to handle more items. The associated hash function must change as the table grows.

The ***load factor*** of a hash table is the ratio of the number of keys in the table to the size of the hash table.

- The higher the load factor, the slower the retrieval.
- With open addressing, the load factor cannot exceed 1.
- With chaining, the load factor often exceeds 1.

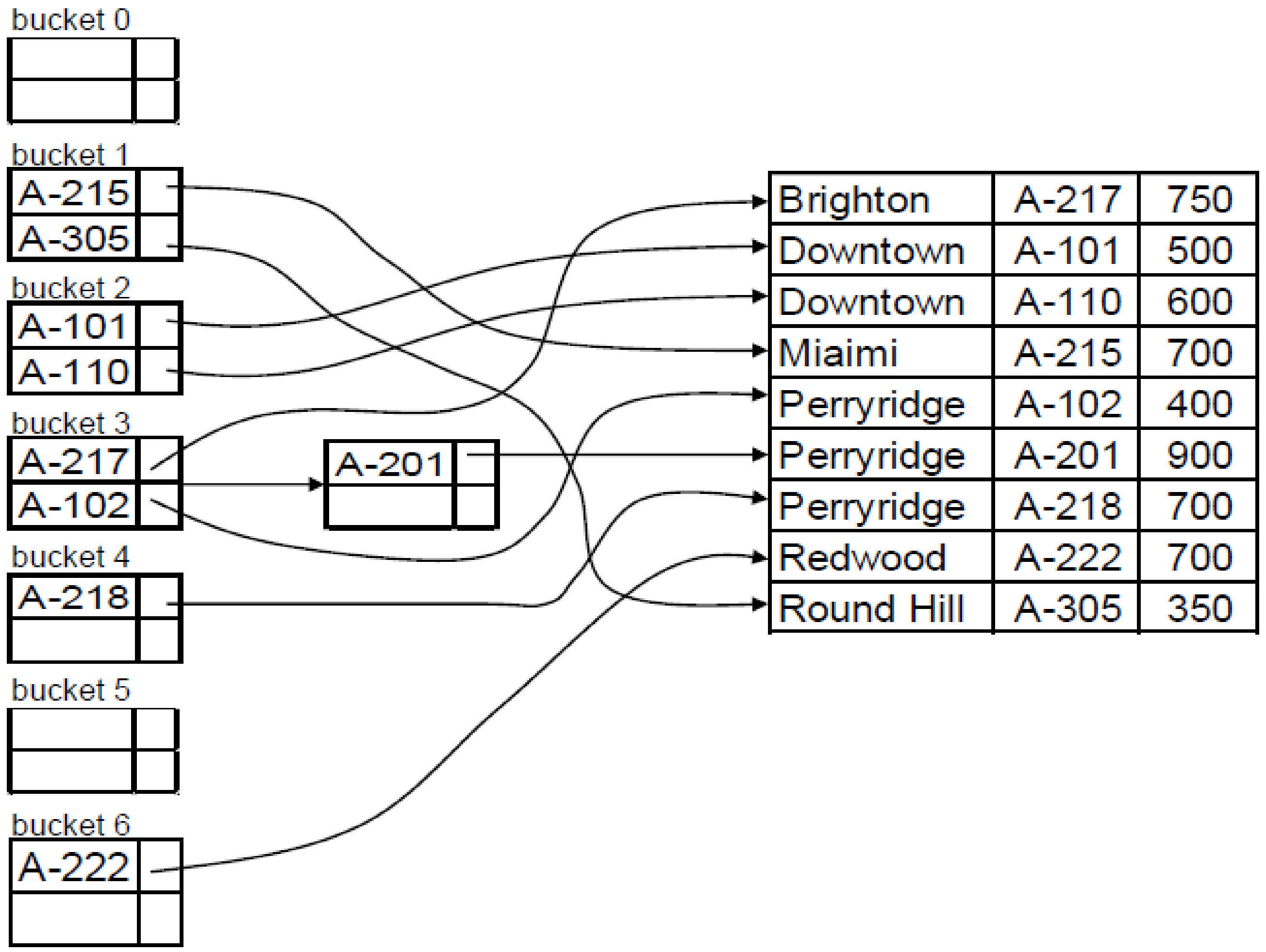
- Open Hashing (Separate Chaining):
- In open hashing, keys are stored in linked lists attached to cells of a hash table.

- Closed Hashing (Open Addressing):
- In closed hashing, all keys are stored in the hash table itself without the use of linked lists.
    - Rehashing

Hash Indices

- Hashing can be used not only for file organization,
- but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Hash indices are always secondary indices

Example of Hash Index



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 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 number of buckets allocated accordingly, 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.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
- splits and coalesces buckets appropriately with the database size.

– i.e. buckets are added and deleted on demand.

.

Dynamic 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 $i = 0$

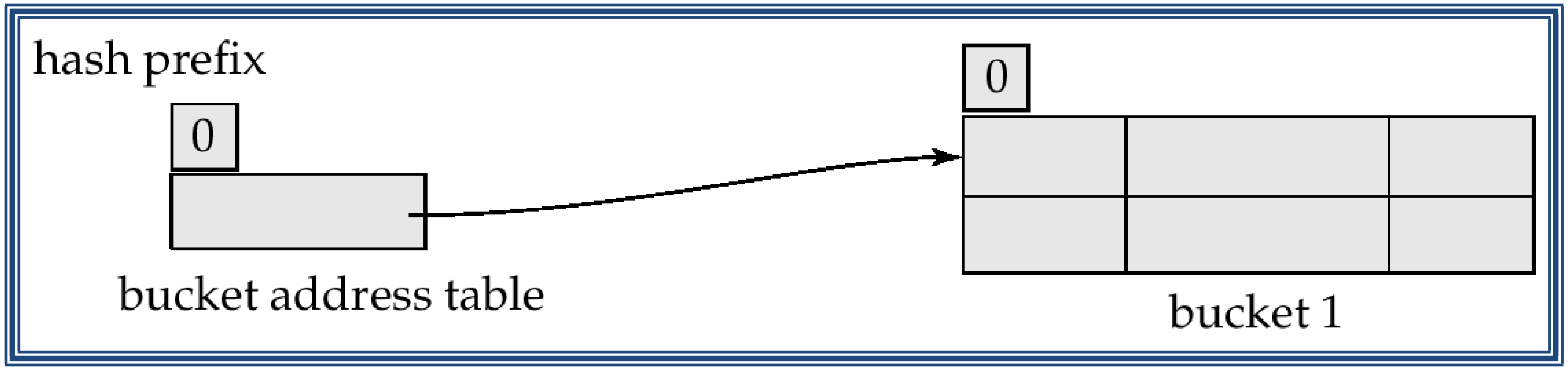
– Value of *i* grows and shrinks as the size of the database grows and shrinks.

– Multiple entries in the bucket address table may point to
a bucket.

– Thus, actual number of buckets is $< 2^{i}$

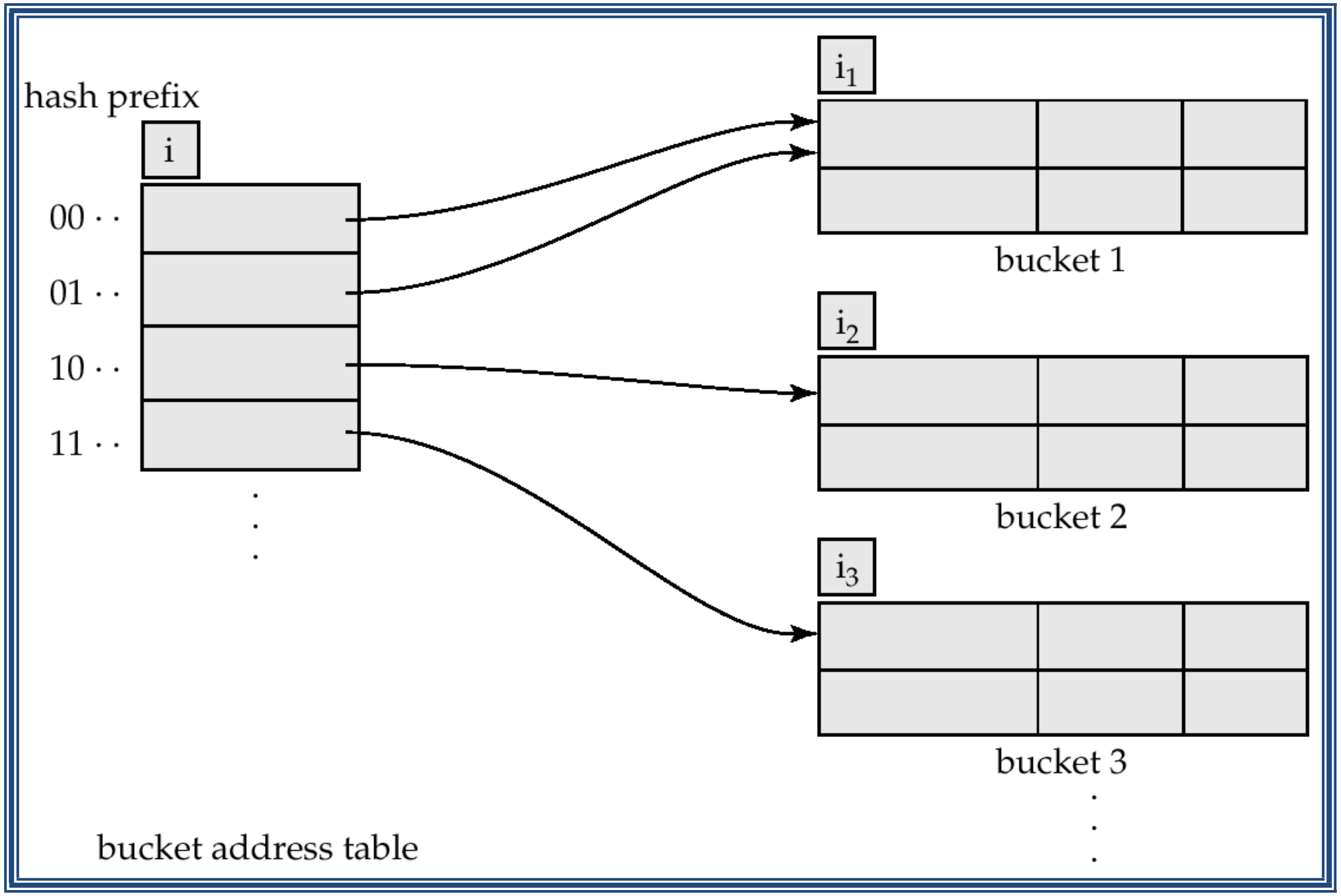
Use of Extendable Hash Structure:

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



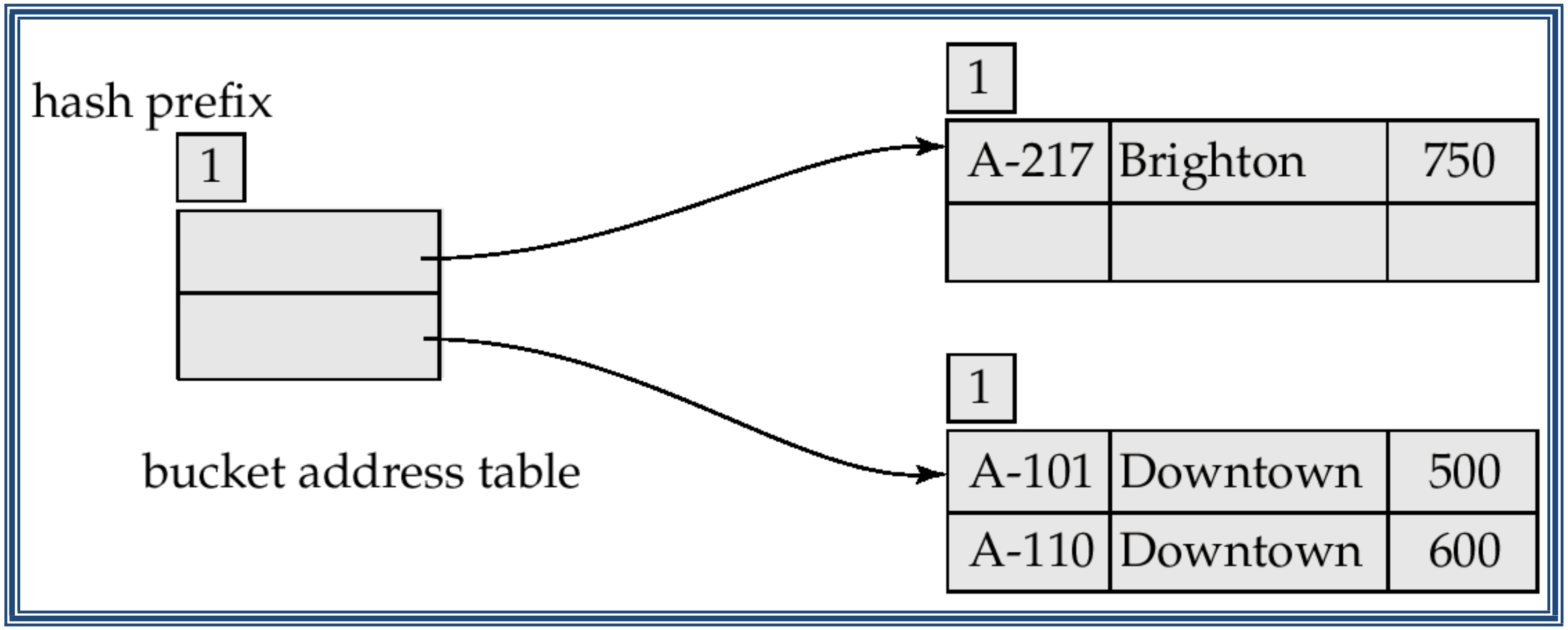
Initial Hash structure, bucket size = 2

Dynamic Hashing



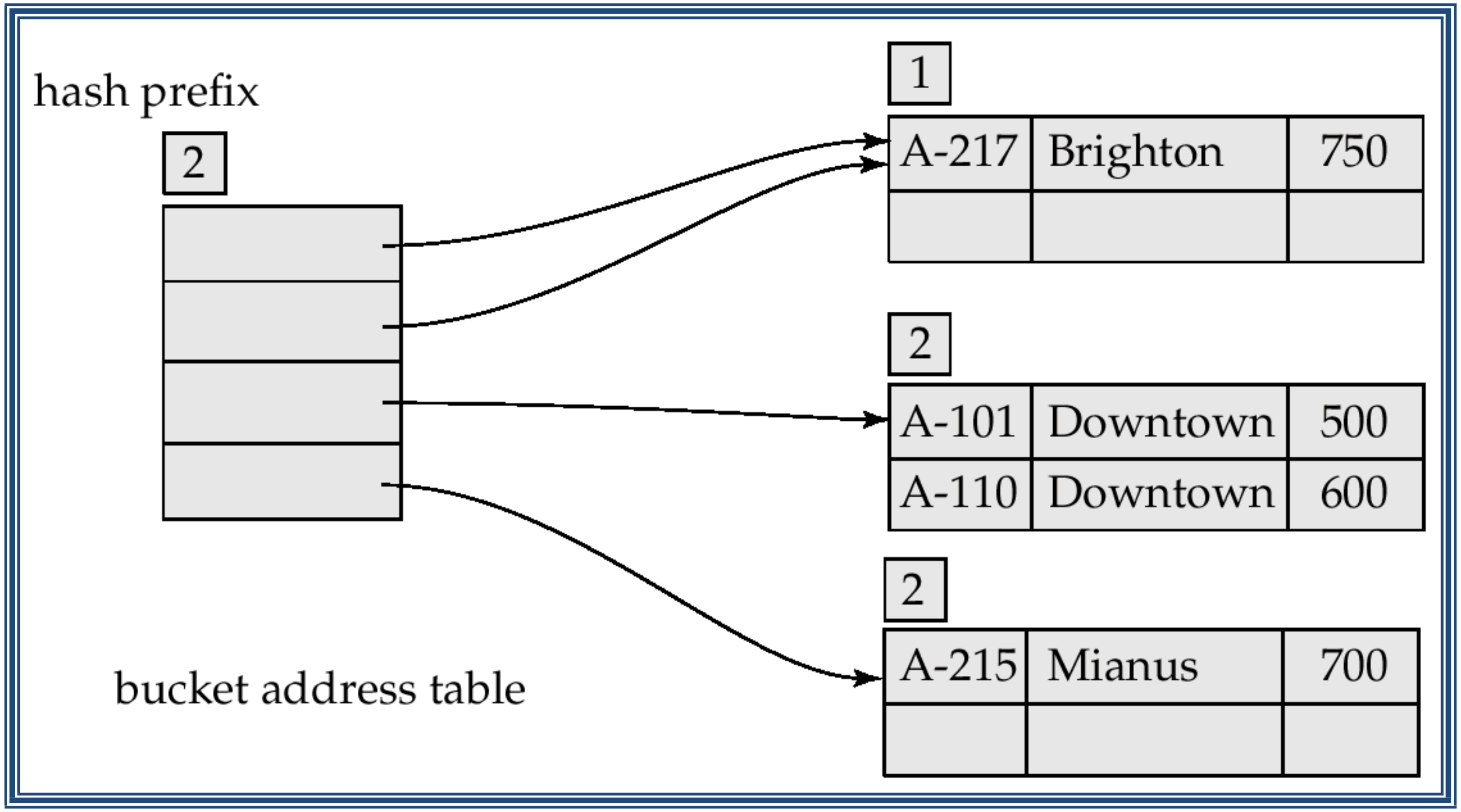
Example (Cont.)

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

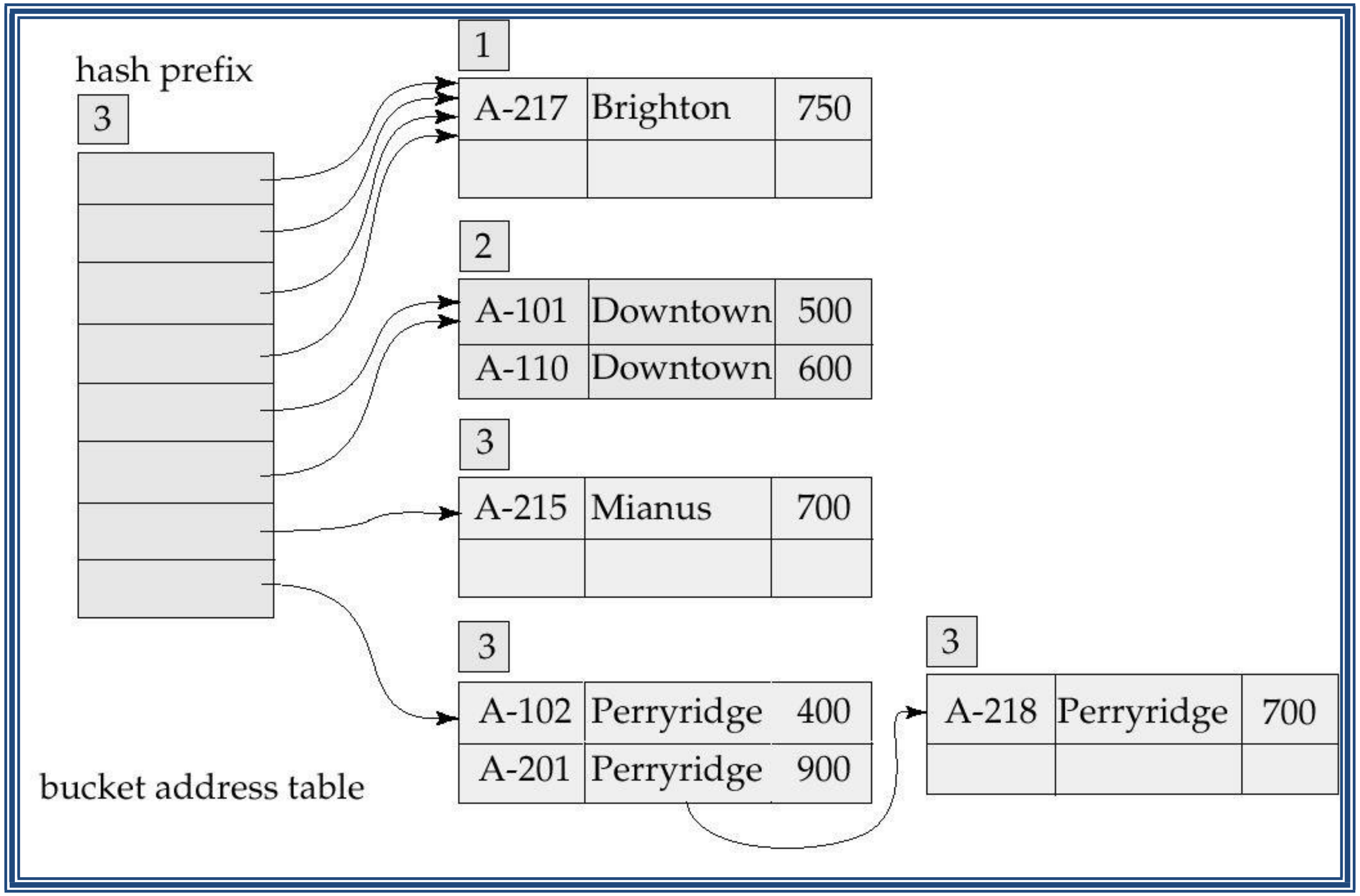


Example (Cont.)

Hash structure after insertion of Mianus record



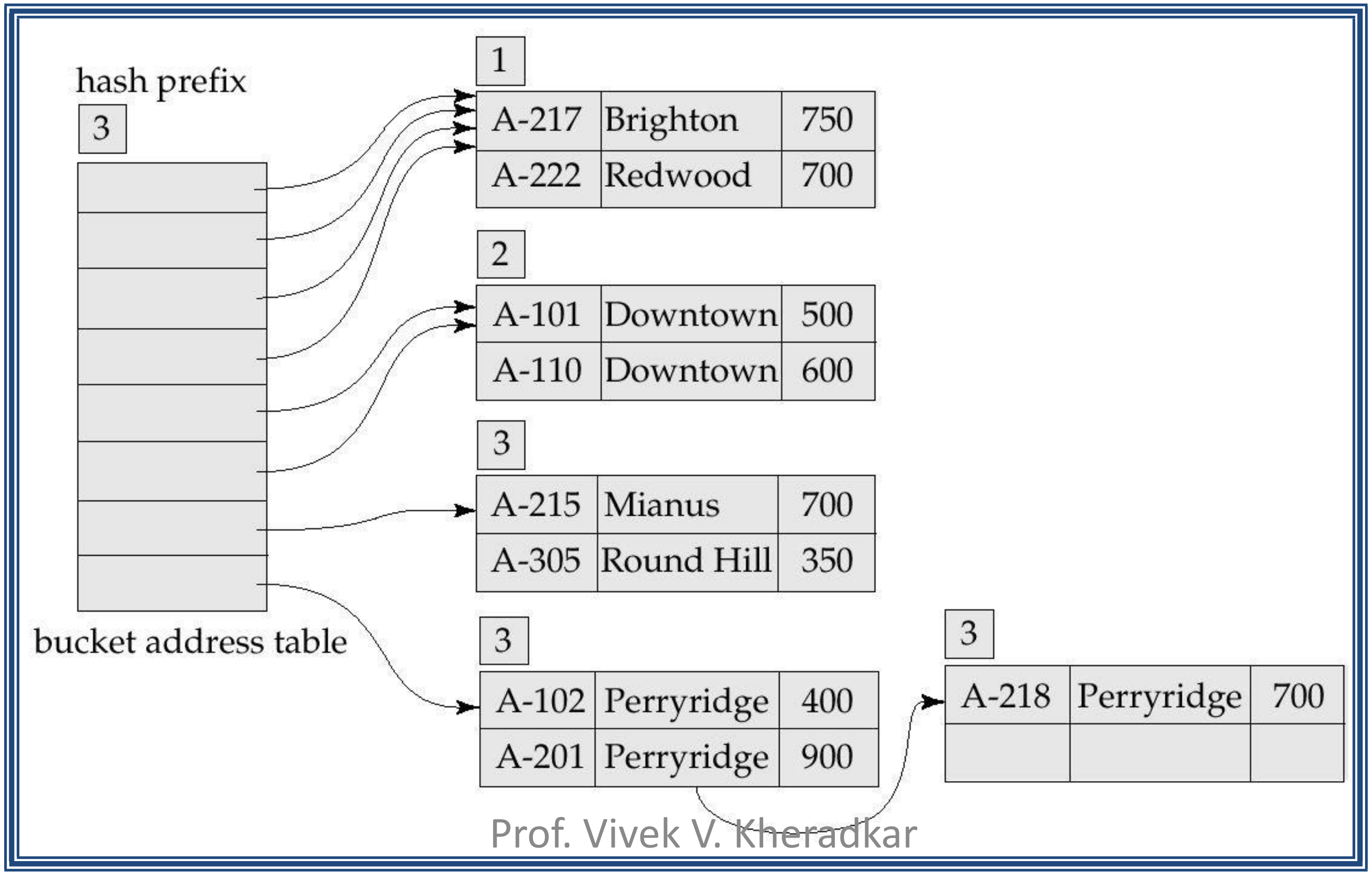
Example (Cont.)



Hash structure after insertion of three Perryridge records

Example (Cont.)

- Hash structure after insertion of Redwood and Round Hill records



Comparison to Other Hashing Methods

- Advantage: performance does not decrease as the database size increases
  - Space is conserved by adding and removing as necessary
- Disadvantage: additional level of indirection for operations
  - Complex implementation

Comparison of Ordered Indexing and Hashing

Issues to consider:

•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