

Module 4

13-10-2021

RAID

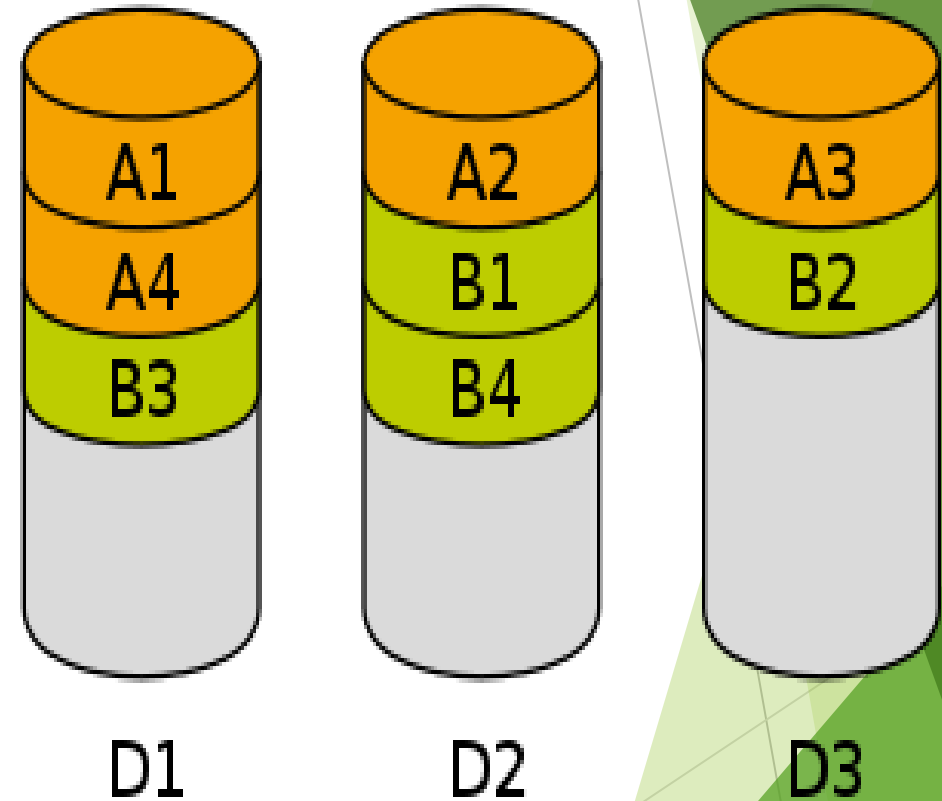
- ▶ RAID or **R**edundant **A**rray of **I**ndependent **D**isks, is a technology to connect multiple secondary storage devices and use them as a single storage media.
- ▶ Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
 - ❖ high capacity and high speed by using multiple disks in parallel, and
 - ❖ high reliability by storing data redundantly, so that data can be recovered even if a disk fails

Improvement of Reliability via Redundancy

- ▶ The solution to the problem of reliability is to introduce redundancy.
- ▶ The simplest approach to introducing redundancy is to duplicate every disk. This technique is called mirroring .
- ▶ A logical disk then consists of two physical disks, and every write is carried out on both disks.
- ▶ If one of the disks fails, the data can be read from the other.
- ▶ Data will be lost only if the second disk fails before the first failed disk is repaired.

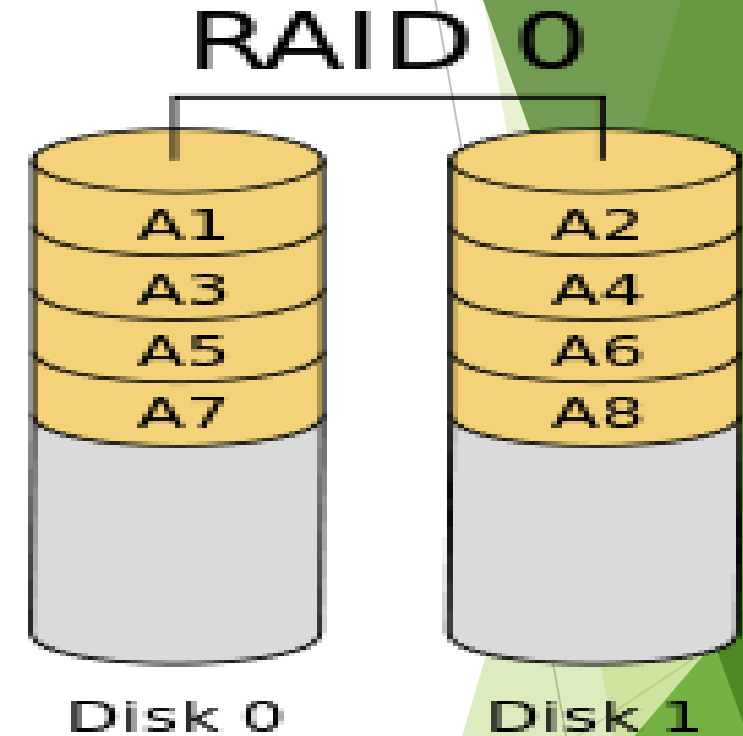
Improvement in performance via Parallelism

- ▶ **Data Striping:** With multiple disks, we can improve the transfer rate by **striping data across multiple disks**
- ▶ **Data striping** is the technique of segmenting logically sequential data
- ▶ RAID consists of 7 levels 0-6



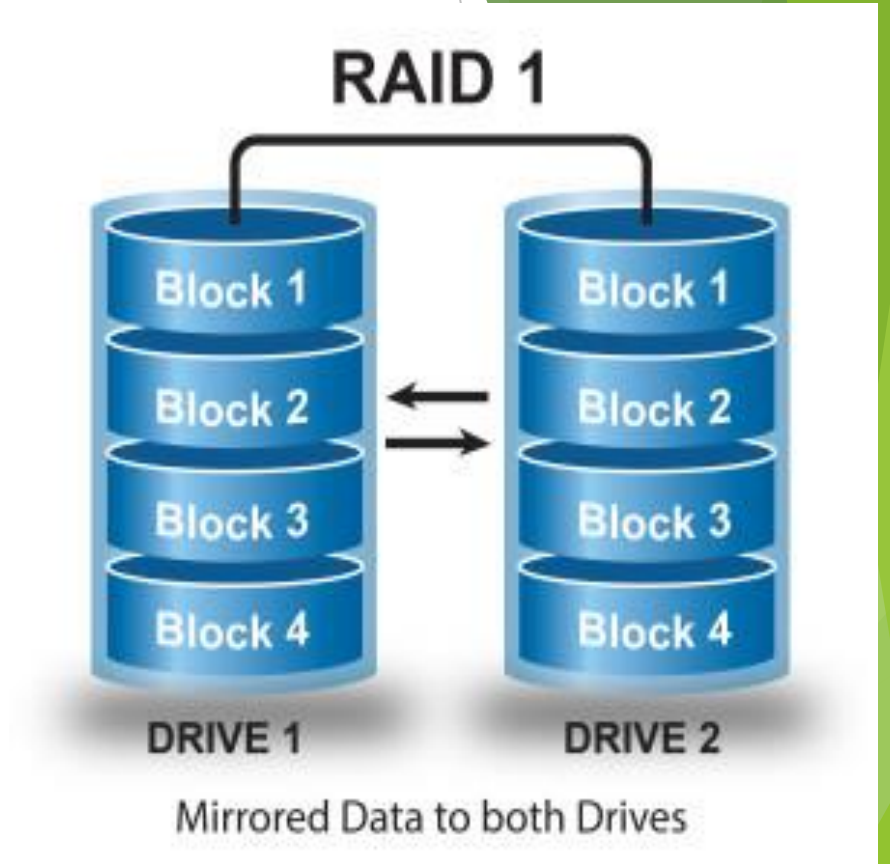
RAID LEVEL 0

- ▶ 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 data to write/read in parallel.
- ▶ It enhances the speed and performance of the storage device. There is no parity and backup in Level 0



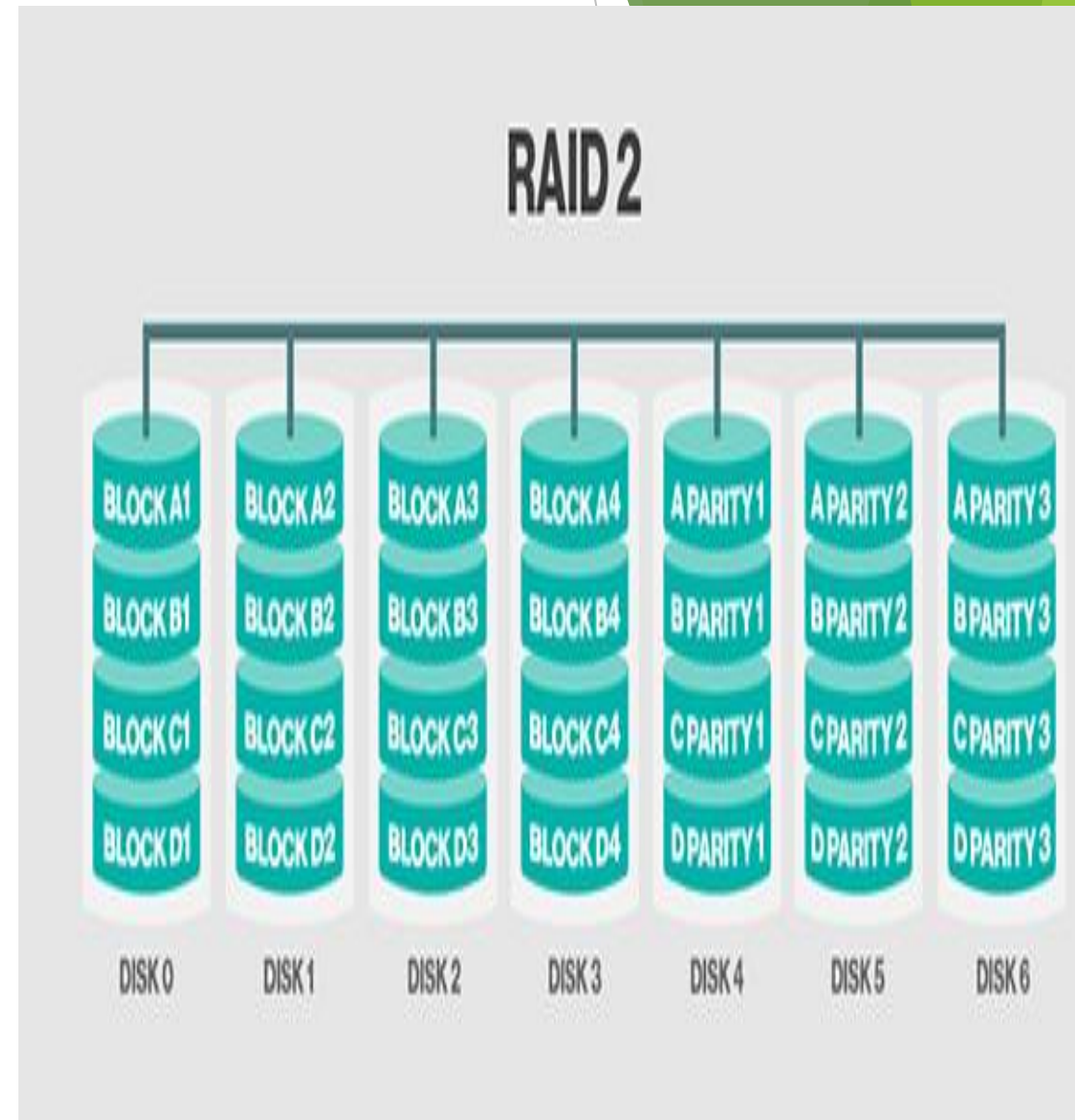
RAID LEVEL 1

- ▶ 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



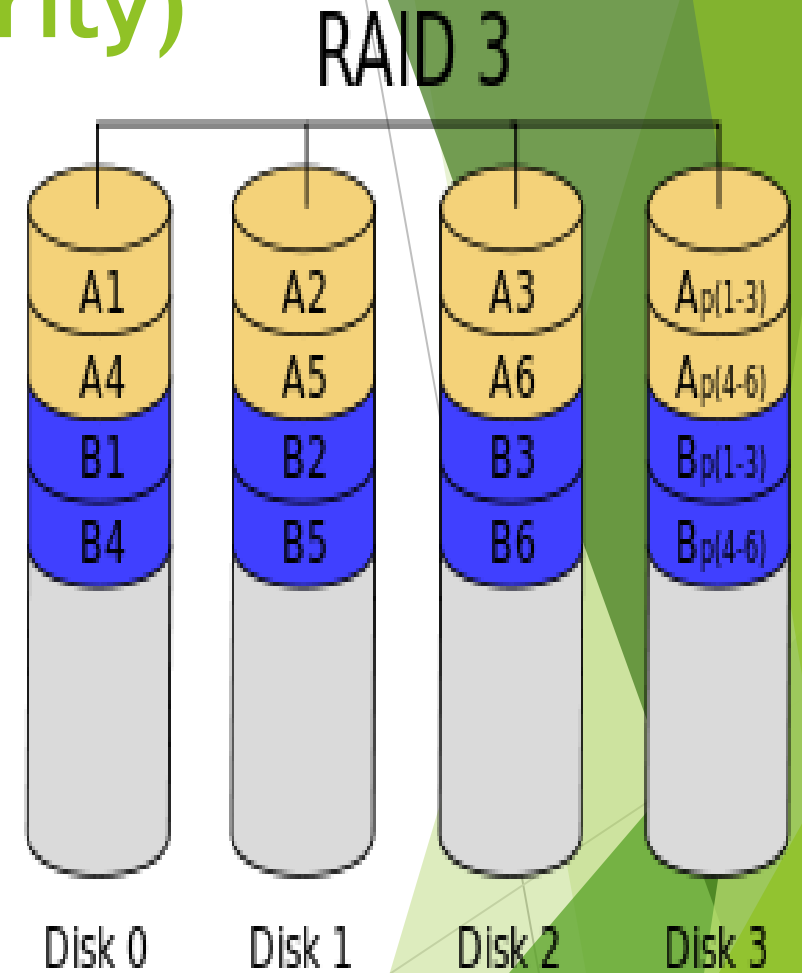
RAID LEVEL 2(Memory Style Error Correcting Code)

- ▶ 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
- ▶ Each byte in a memory system may have a parity bit associated with it that records whether the number of bits in that byte that are set to 1 is even or odd



RAID LEVEL 3(bit interleaved parity)

- ▶ RAID level 3 can detect whether a sector has been read correctly,so a single parity bit can be used for error correction as well as for detection
- ▶ The idea is as follows,if one of the sector get damaged,system exactly know which sector it is and for each bit in the sector ,the system can figure out whether it is a 1 or 0 by computing the parity of the corresponding bits from sectors in other disk
- ▶ If the parity of remaining bit is equal to stored parity,the missing bit is 0,otherwise 1
- ▶ Parity value is XOR value of data bits

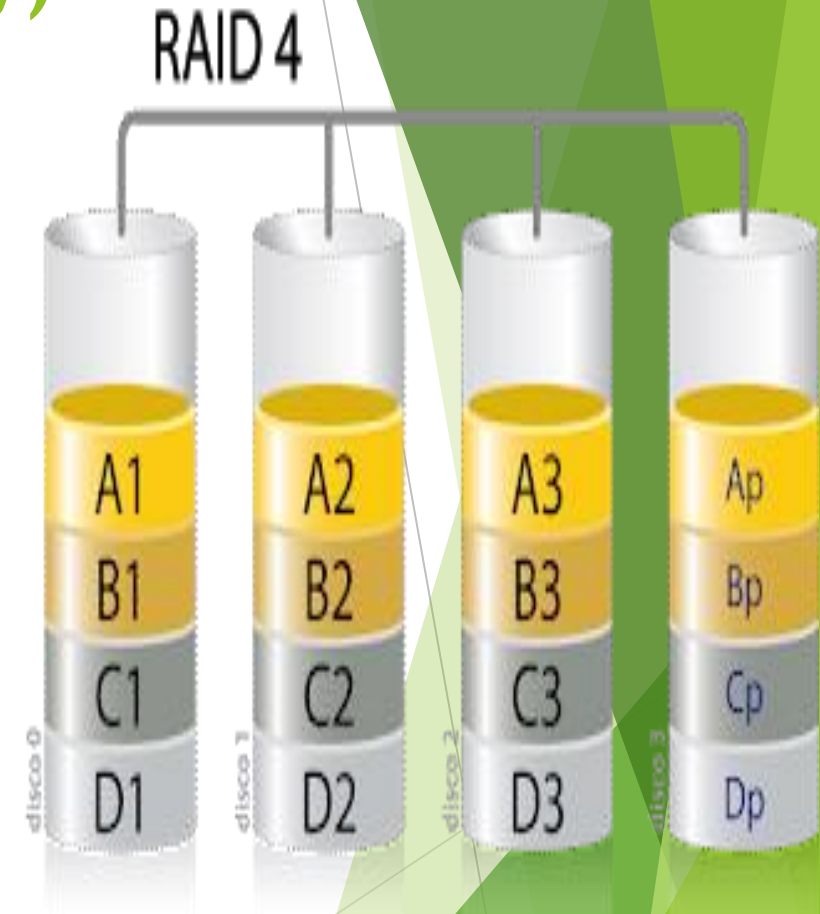


RAID LEVEL 4(block interleaved parity)

- ▶ Uses block level stripping and keeps a parity block on a separate disk for corresponding blocks from N other disks
- ▶ If one of the disk fails,parity block can be used with corresponding blocks from other disks to restore the blocks of failed disk
- ▶ A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate.
- ▶ The transfer rates for large reads is high, since all the disks can be read in parallel;large writes also have high transfer rates, since the data and parity can be written in parallel.

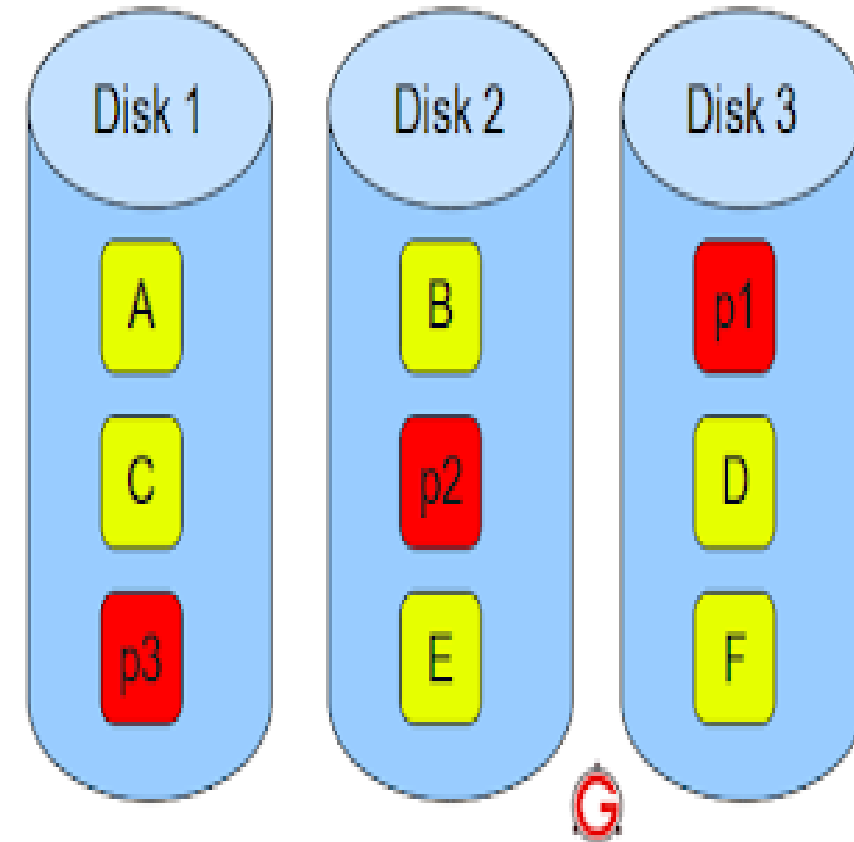
RAID LEVEL 4(block interleaved parity)

- ▶ A write of a block has to access the disk on which the block is stored, as well as the parity disk, since the parity block has to be updated.
- ▶ Moreover, both the old value of the parity block and the old value of the block being written have to be read for the new parity to be computed.
- ▶ Thus, a single write requires four disk accesses: two to read the two old blocks, and two to write the two blocks



RAID LEVEL 5(block interleaved distributed parity)

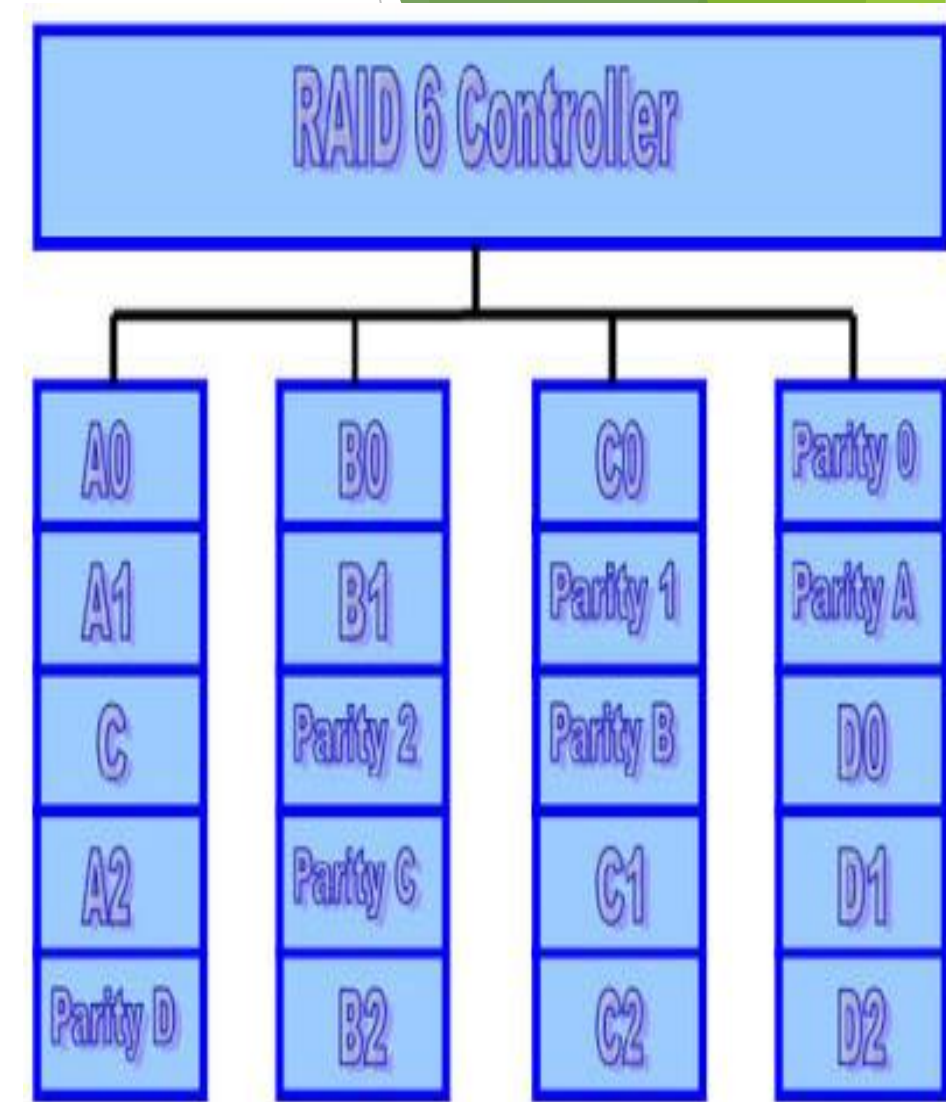
- ▶ RAID Level 5 improves upon Level 4 by distributing the parity blocks uniformly over all disks, instead of storing them as a single check disk.
- ▶ This distribution has two advantages. First, several write requests can potentially be processed in parallel, since the bottleneck of a unique check disk has been eliminated.
- ▶ Second, read requests have a higher level of parallelism



RAID 5 – Blocks Striped. Distributed Parity.

RAID LEVEL 6(P+Q redundancy Scheme)

- ▶ A RAID Level 6 system uses Reed - Solomon codes to be able to recover from upto two simultaneous disk failure
- ▶ Stores extra redundant information to guard against multiple disk failures



Choice of RAID Level

The factors to be taken into account in choosing a RAID level are:

- Cost of extra disk-storage requirements.
- ▶ Performance requirements in terms of number of I/O operations.
- ▶ Performance when a disk has failed.
- ▶ Performance during rebuild (that is, while the data in a failed disk are being rebuilt on a new disk).

Choice of RAID Level

- ▶ The time to rebuild the data of a failed disk can be significant, and it varies with the RAID level that is used.
- ▶ Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data of a failed disk.
- ▶ Bit striping (level 3) is inferior to block striping (level 5), since block striping gives as good data-transfer rates for large transfers, while using fewer disks for small transfers
- ▶ The choice between RAID level 1 and level 5 is harder to make.
- ▶ RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance.
- ▶ RAID level 5 has a lower storage overhead than level 1, but has a higher time overhead for writes. For applications where data are read frequently, and written rarely, level 5 is the preferred choice

Choice of RAID Level

- ▶ The time to rebuild the data of a failed disk can be significant, and it varies with the RAID level that is used.
- ▶ Rebuilding is easiest for RAID level 1, since data can be copied from another disk; for the other levels, we need to access all the other disks in the array to rebuild data of a failed disk.
- ▶ Bit striping (level 3) is inferior to block striping (level 5), since block striping gives as good data-transfer rates for large transfers, while using fewer disks for small transfers
- ▶ The choice between RAID level 1 and level 5 is harder to make.
- ▶ RAID level 1 is popular for applications such as storage of log files in a database system, since it offers the best write performance.
- ▶ RAID level 5 has a lower storage overhead than level 1, but has a higher time overhead for writes. For applications where data are read frequently, and written rarely, level 5 is the preferred choice

File Organization

- ▶ A file is organized logically as a sequence of records.
- ▶ These records are mapped onto disk blocks. Each file is also logically partitioned into fixed-length storage units called blocks, which are the units of both storage allocation and data transfer.
- ▶ A block may contain several records. In addition, we shall require that each record is entirely contained in a single block; that is, no record is contained partly in one block, and partly in another

Fixed Length Records

- ▶ Instead of allocating a variable amount of bytes for the attributes, we allocate the maximum number of bytes that each attribute can hold.
- ▶ As an example, let us consider a file of instructor records for our university database. Each record of this file is defined (in pseudocode) as:

```
type instructor = record  
    ID varchar (5);  
    name varchar(20);  
    dept_name varchar (20);  
    salary numeric (8,2);  
end
```

- ▶ The instructor record is 53 bytes long. A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on.

Fixed Length Records

- ▶ There are two problems with this approach
 - ❑ It is difficult to delete a record from this structure. The space occupied by record to be deleted must be filled with some other record of the file
 - ❑ Unless the block size happens to be a multiple of 53, some records will cross block boundaries. That is, part of the record will be stored in one block and part in another
 - ❑ It would thus require two block accesses to read or write such a record

record 0

record 1

record 2

record 3

record 4

record 5

record 6

record 7

record 8

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

Fixed Length Records

- When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead.
- It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Thus, we need to introduce an additional structure

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

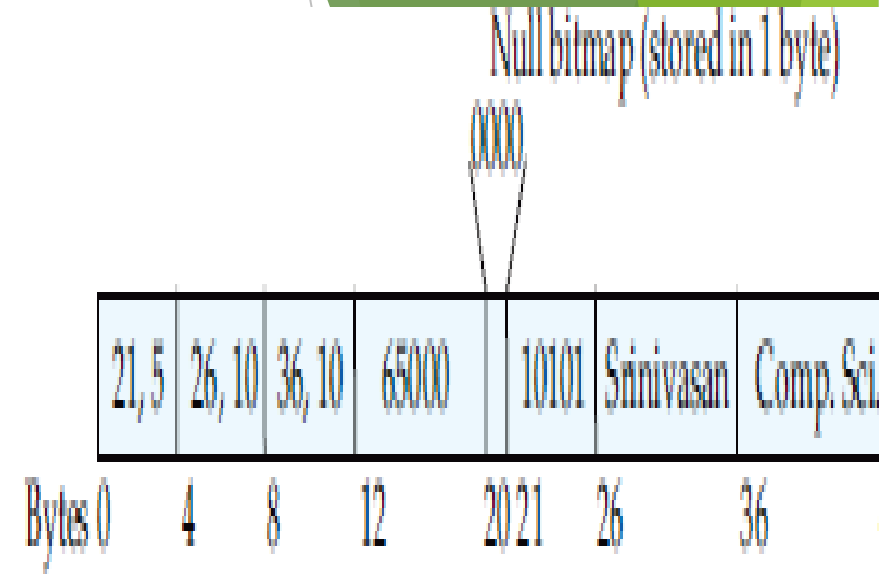
Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they “point” to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers.
- (No pointers stored in use records.)

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

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, such as arrays or multi sets.
- ▶ The representation of a record with variable-length attributes typically has two parts:
 - an initial part with fixed length attribute
 - data for variable length attributes



Representation of variable-length record.

Variable Length Records

- ▶ Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value.
- ▶ Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (offset, length), where offset denotes where the data for that attribute begins within the record, and length is the length in bytes of the variable-sized attribute
- ▶ The values for these attributes are stored consecutively, after the initial fixed-length part of the record
- ▶ Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length

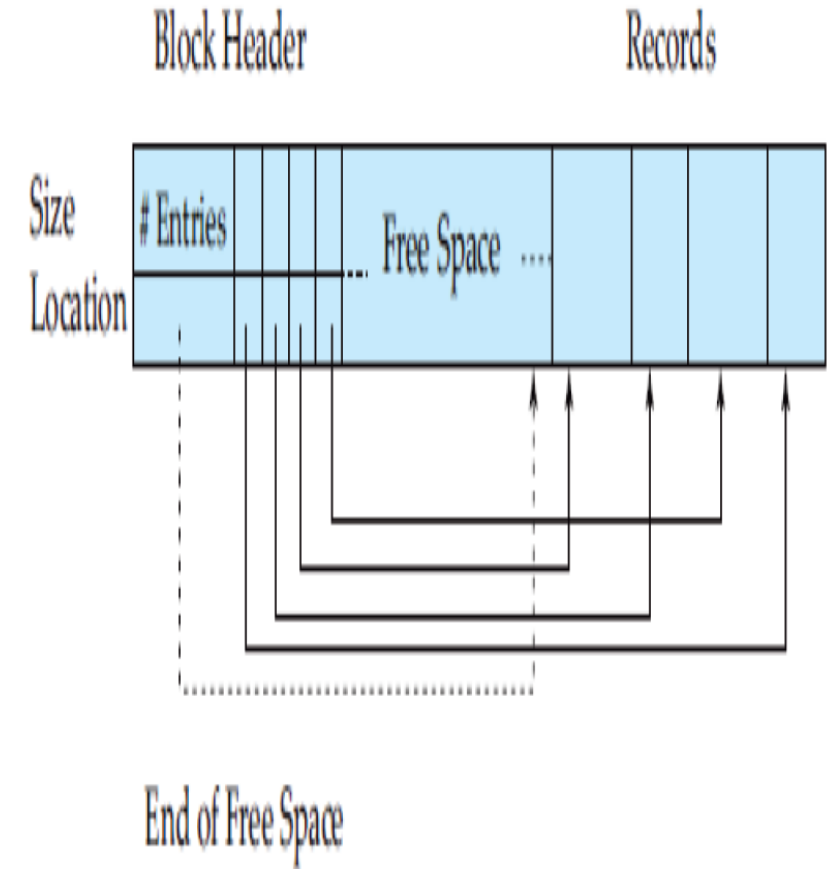
Variable Length Records

- ▶ The figure shows an instructor record, whose first three attributes ID, name, and dept name are variable-length strings, and whose fourth attribute salary is a fixed-sized number.
- ▶ The figure also illustrates the use of a null bitmap, which indicates which attributes of the record have a null value.
- ▶ In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored in bytes 12 through 19 would be ignored

Variable Length Records

Slotted page structure

- The slotted-page structure is commonly used for organizing records within a block
- ▶ There is a header at the beginning of each block, containing the following information:
 1. The number of record entries in the header.
 2. The end of free space in the block.
 3. An array whose entries contain the location and size of each record
- The actual records are allocated contiguously in the block, starting from the end of the block.
- The free space in the block is contiguous, between the final entry in the header array, and the first record.



Slotted-page structure.

Variable Length Records

Slotted page structure

- If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.
- If a record is deleted, the space that it occupies is freed, and its entry is set to deleted (its size is set to -1 , for example).
- Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the header array and the first record.
- The end-of-free-space pointer in the header is appropriately updated as well.

Organization of Records in Files

- Several of the 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. There is no ordering of records. Typically, there is a single file for each relation.
- ▶ 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; it need not be the primary key, or even a super key.
- To permit fast retrieval of records in search-key order, we chain together records by pointers.
- The pointer in each record points to the next record in search-key order. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible

Sequential File Organization

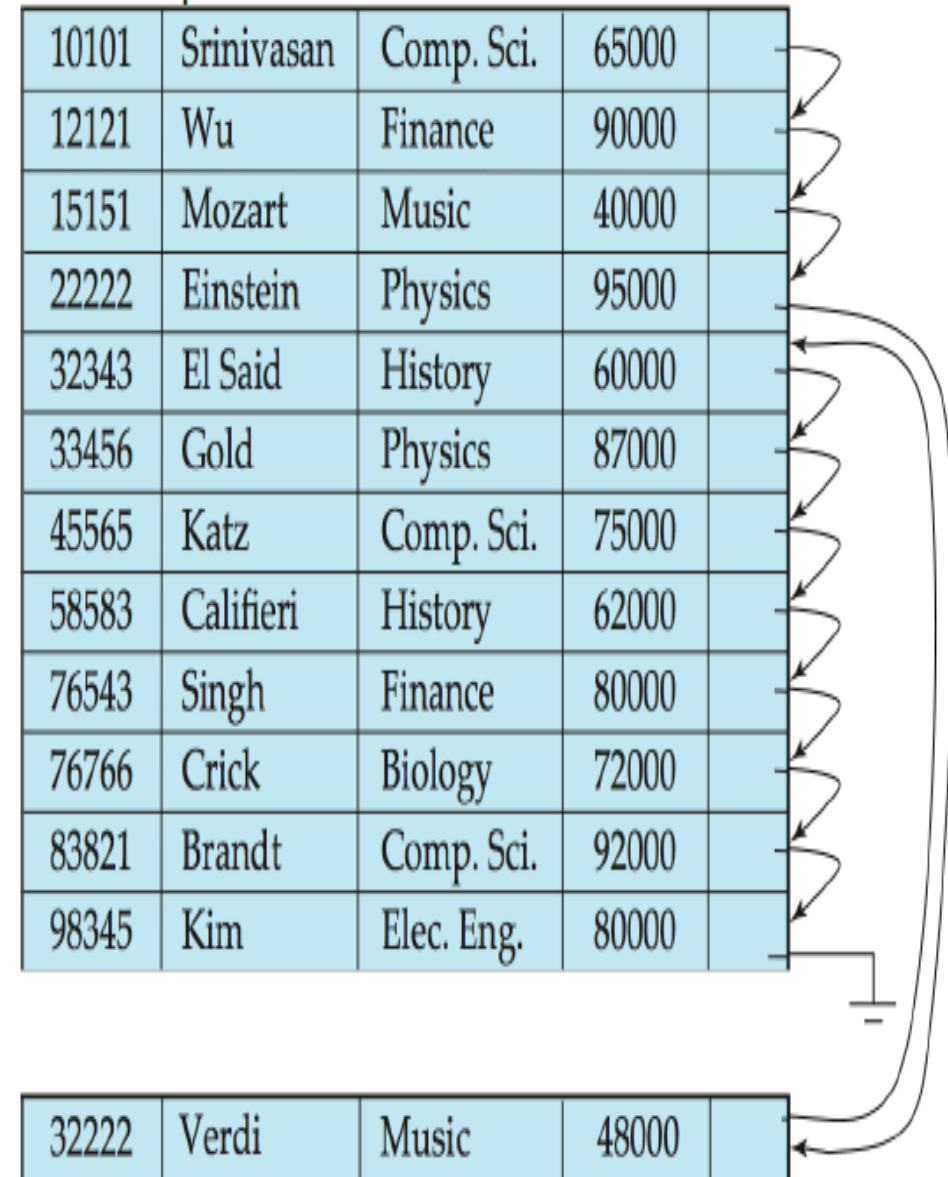
- ▶ Figure shows a sequential file of instructor records taken from our university example.
- ▶ In that example, the records are stored in search-key order, using ID as the search key.
- ▶ The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms.

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



Sequential File Organization

- Insertion & Deletion :
- Deletion is performed using pointer chains
- For insertion, we apply the following rules:
 1. Locate the record in the file that comes before the record to be inserted in search-key order.
 2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search-key order.



Sequential File Organization

► Advantages

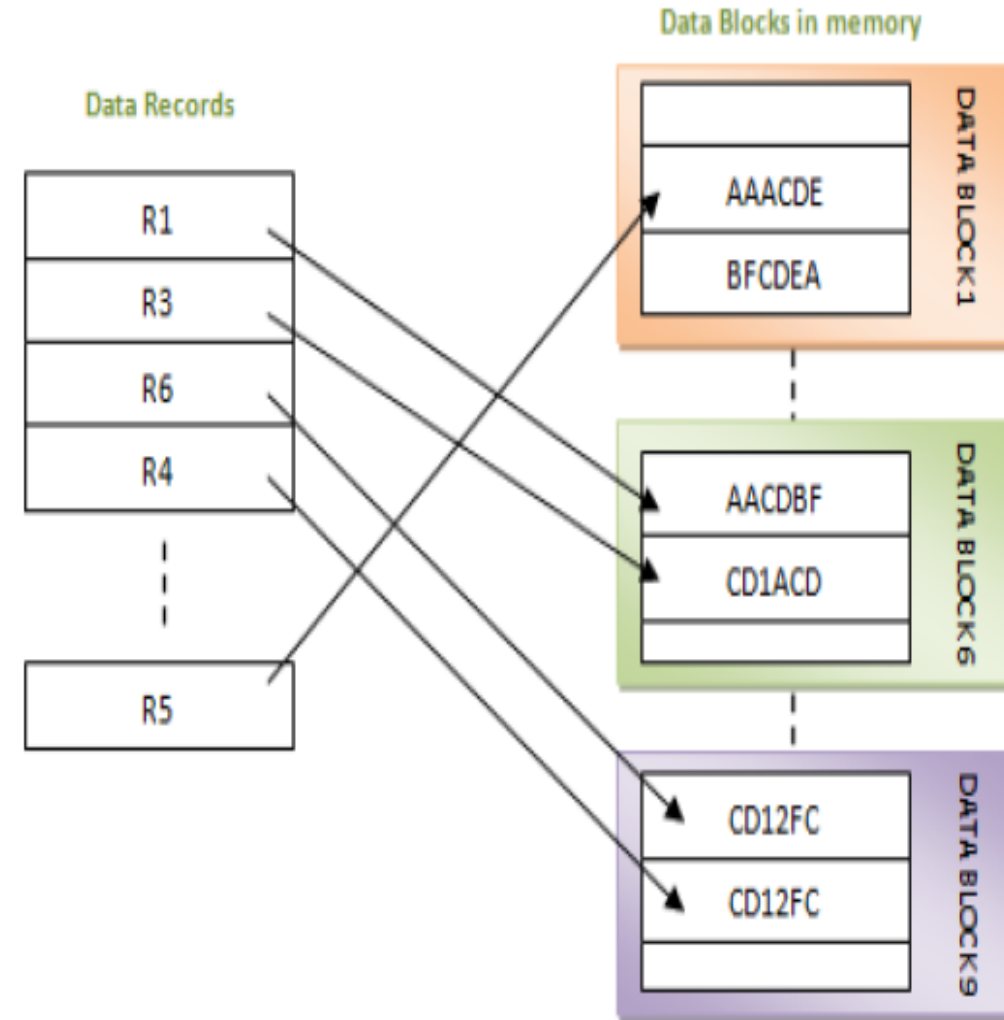
- The design is very simple compared other file organization. There is no much effort involved to store the data.
- This method is helpful when most of the records have to be accessed like calculating the grade of a student, generating the salary slips etc where we use all the records for our calculations
- This method is good in case of report generation or statistical calculations.
- These files can be stored in magnetic tapes which are comparatively cheap

► Disadvantages

- Sorted file method always involves the effort for sorting the record.
- Each time any insert/update/ delete transaction is performed, file is sorted.Hence identifying the record, inserting/ updating/ deleting the record, and then sorting them always takes some time and may make system slow.

Heap File Organization

- ▶ This is the simplest form of file organization.
- ▶ Records are inserted at the end of the file as and when they are inserted.
- ▶ There is no sorting or ordering of the records.
- ▶ Once the data block is full, the next record is stored in the new block.
- ▶ This new block need not be the very next block.
- ▶ This method can select any block in the memory to store the new records.
- ▶ DBMS is responsible to store and manage the records



Heap File Organization

- ▶ When a record has to be retrieved from the database, in this method, we need to traverse from the beginning of the file till we get the requested record.
- ▶ Hence fetching the records in very huge tables, it is time consuming. This is because there is no sorting or ordering of the records. We need to check all the data.
- ▶ Similarly if we want to delete or update a record, first we need to search for the record. Again, searching a record is similar to retrieving it- start from the beginning of the file till the record is fetched.
- ▶ While deleting a record, the record will be deleted from the data block. But it will not be freed and it cannot be re-used.
- ▶ Hence as the number of record increases, the memory size also increases.
- ▶ For the database to perform better, DBA has to free this unused memory periodically.

Heap File Organization

Advantages

- ▶ Very good method of file organization for bulk insertion.
- ▶ when there is a huge number of data needs to load into the database at a time, then this method of file organization is best suited. They are simply inserted one after the other in the memory blocks.
- ▶ It is suited for very small files as the fetching of records is faster in them. As the file size grows, linear search for the record becomes time consuming.

Disadvantages

- ▶ This method is inefficient for larger databases as it takes time to search/modify the record.
- ▶ Proper memory management is required to boost the performance. Otherwise there would be lots of unused memory blocks lying and memory size will simply be growing.

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.
- ▶ Use the term bucket to denote 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.
- ▶ Formally, 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 perform a lookup on a search-key value K_i , we simply compute $h(K_i)$, then search the bucket with that address.

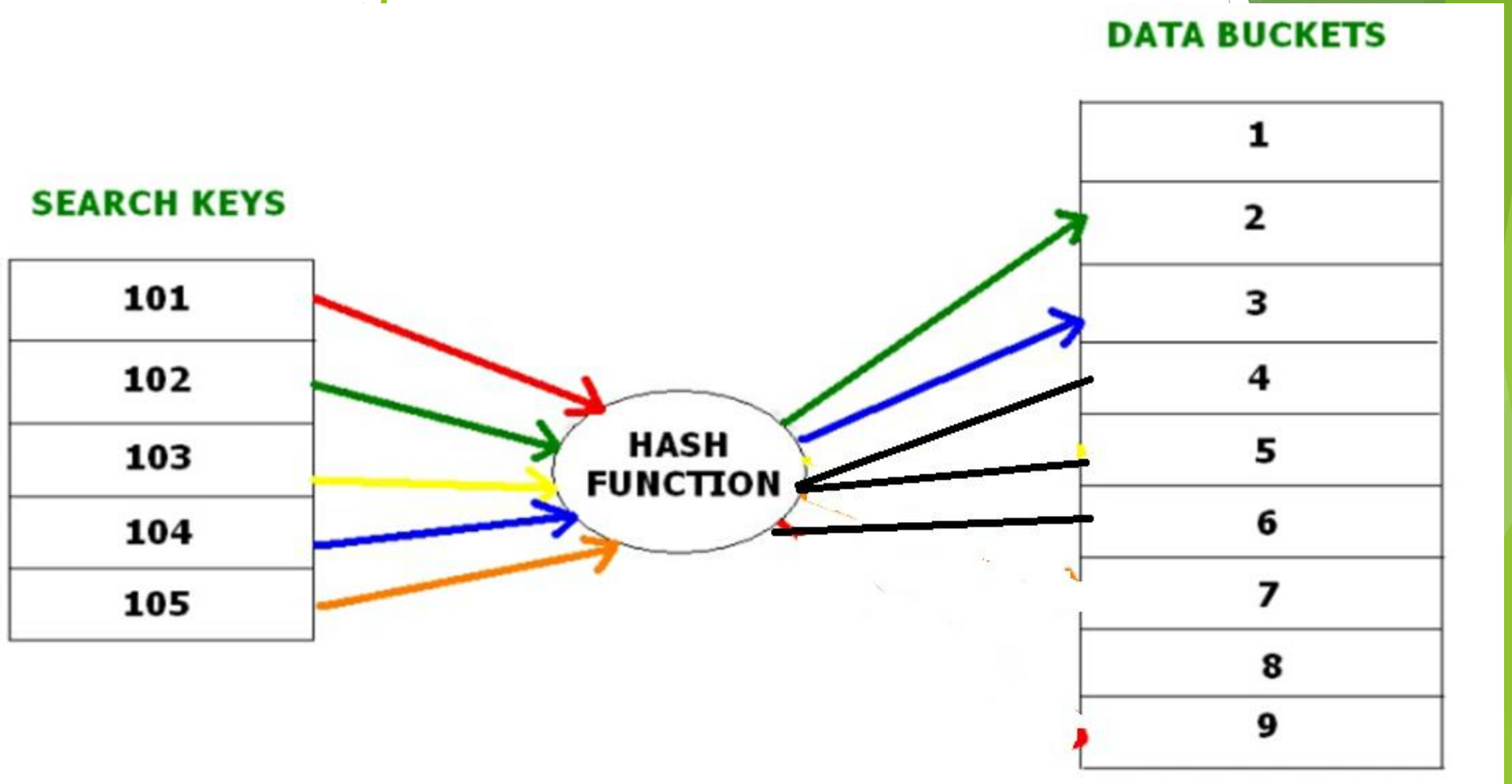
Hash File Organization

- ▶ Suppose that two search keys, K5 and K7, have the same hash value; that is, $h(K5) = h(K7)$.
- ▶ If we perform a lookup 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
- ▶ To insert a record with search key K_i , we compute $h(K_i)$, which gives the address of the bucket for that record.
- ▶ Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket
- ▶ Deletion is equally straightforward. If the search-key value of the record to be deleted is K_i , we compute $h(K_i)$, then search the corresponding bucket for that record, and delete the record from the bucket.

Hash File Organization

- ▶ An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records. we want to 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. That is, 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. That is, 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.
- ▶ Typical hash functions perform computation on the internal binary machine representation of characters in the search key.
- ▶ A simple hash function of this type first computes the sum of the binary representations of the characters of a key, then returns the sum modulo the number of buckets.

Hash File Organization



Example of Hash File Organization

Hash file organization of *account* file, using *branch_name* as key

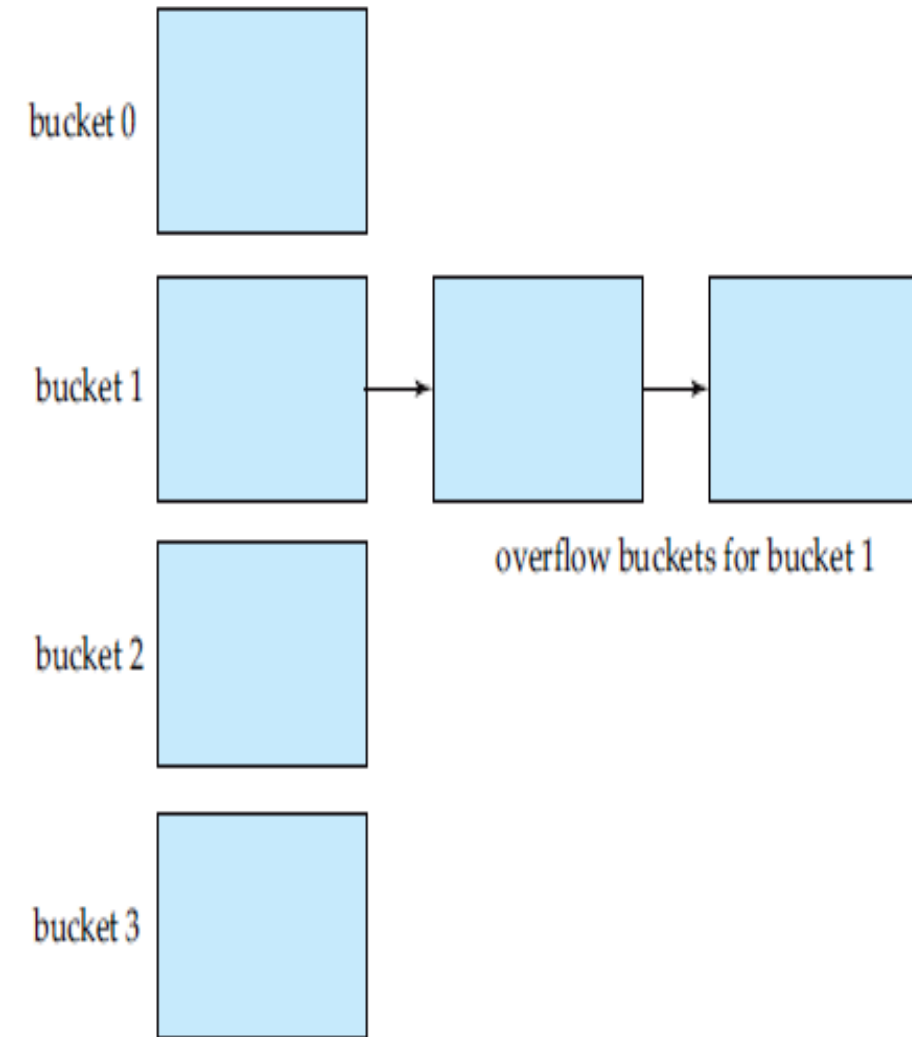
bucket 0			
bucket 1			
bucket 2			
bucket 3	A-217	Brighton	750
	A-305	Round Hill	350
bucket 4	A-222	Redwood	700
bucket 5	A-102	Perryridge	400
	A-201	Perryridge	900
	A-218	Perryridge	700
bucket 6			
bucket 7	A-215	Mianus	700
bucket 8	A-101	Downtown	500
	A-110	Downtown	600
bucket 9			

- There are 10 buckets,
- The binary representation 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.:
 - ▶ $h(\text{Perryridge}) = 5$
 - ▶ $h(\text{Round Hill}) = 3$
 - ▶ $h(\text{Brighton}) = 3$

Hash File Organization

Bucket Overflow

- ▶ The condition of bucket-overflow is known as **collision**.
- ▶ Overflow Chaining – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called Closed Hashing
- ▶ Linear Probing – When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called Open Hashing.



Overflow chaining in a hash structure.

Hash File Organization

Advantages

- ▶ Records need not be sorted after any of the transaction. Hence the effort of sorting is reduced in this method.
- ▶ Since block address is known by hash function, accessing any record is very faster. Similarly updating or deleting a record is also very quick
- ▶ This method can handle multiple transactions as each record is independent of other. i.e.; since there is no dependency on storage location for each record, multiple records can be accessed at the same time

Disadvantages

- ▶ This method may accidentally delete the data.
- ▶ Since all the records are randomly stored, they are scattered in the memory. Hence memory is not efficiently used.

Basic Concepts of Indexing

- ▶ Indexing mechanisms used to speed up access to desired data.
- ▶ – E.g., author catalog in library
- ▶ **Search Key** – an attribute or a set of attributes used to look up records in a file.
- ▶ An **index file** consists of records (called **index entries**) of the form

search-key	pointer
------------	---------

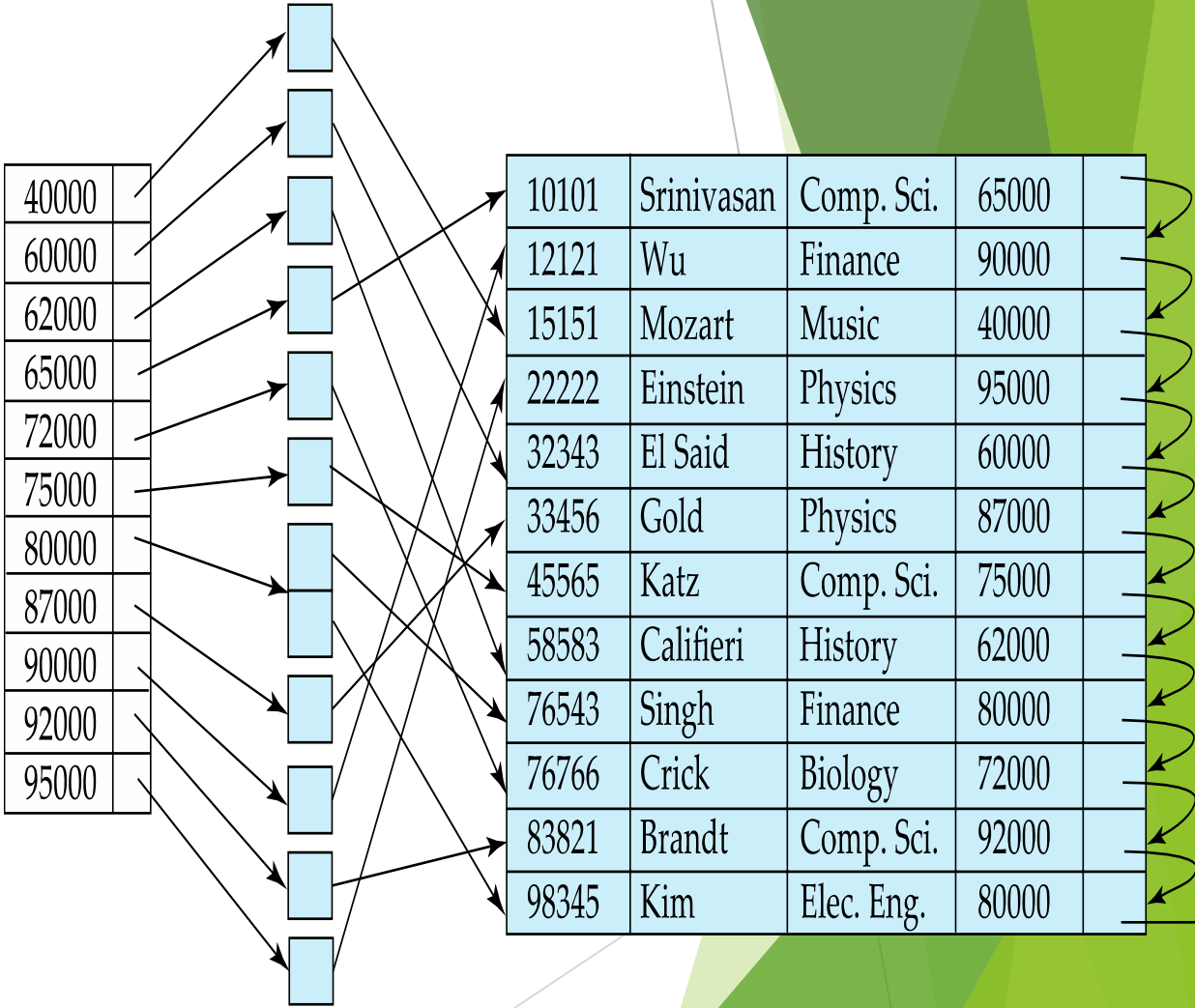
- ▶ Index files are typically much smaller than the original file
- ▶ Two basic kinds of indices:
- ▶ **Ordered indices:** search keys are stored in sorted order
- ▶ **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

Ordered Indices

- ▶ In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- ▶ **Primary index**: In a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - ▶ Also called **clustering index**
 - ▶ The search key of a primary index is usually but not necessarily the primary key.
- ▶ **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- ▶ **Index-sequential file**: ordered sequential file with a primary index.

Ordered Indices

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



Dense & Sparse Indices

- ▶ An **index entry, or index record**, 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 that we can use:
 - ▶ Dense Index
 - ▶ Sparse Index

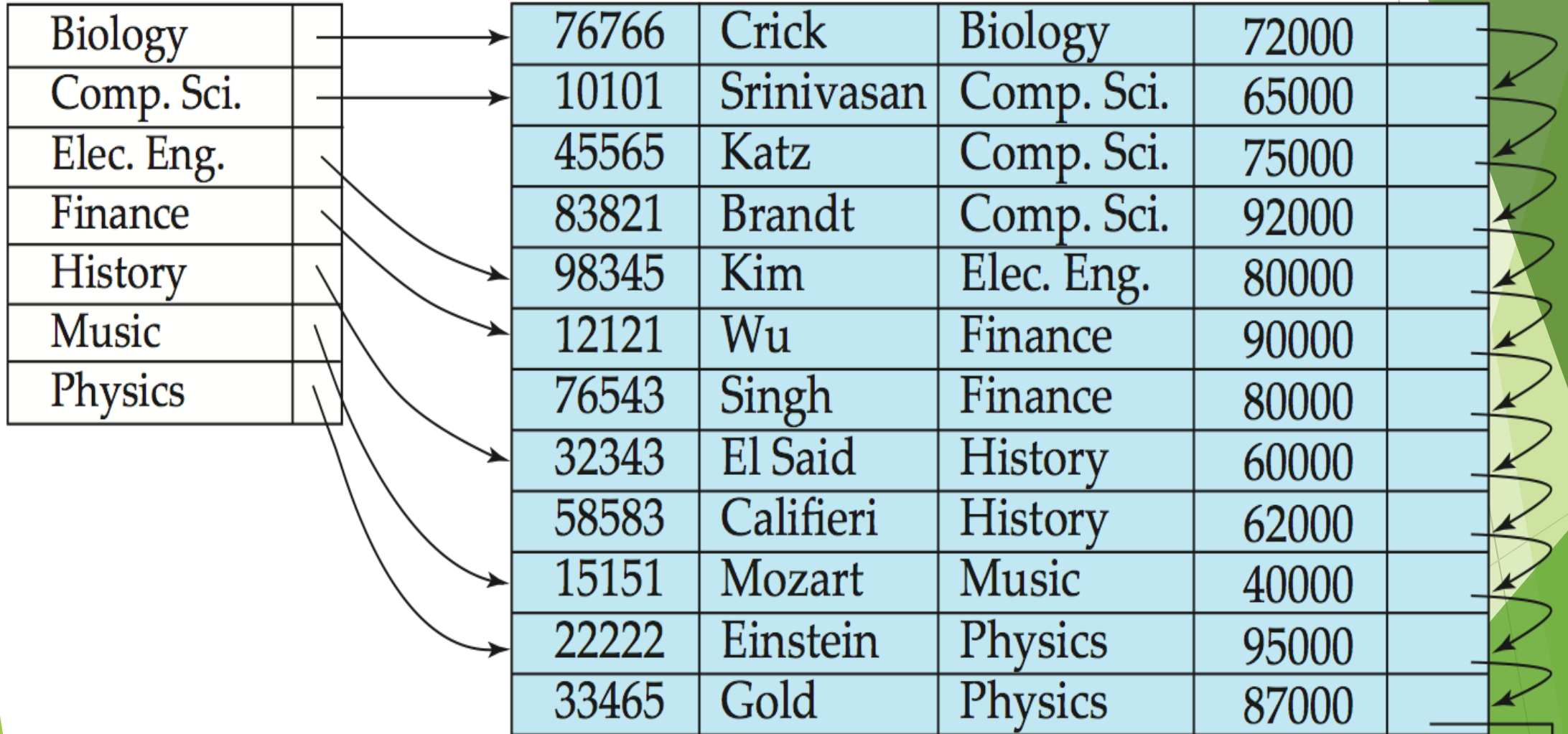
Dense & Sparse Indices

- ▶ **Dense index** — **Index record appears for every search-key value in the file.**
- ▶ E.g. index on *ID attribute of instructor relation*
- ▶ In a dense clustering 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, since, because the index is a clustering one, records are sorted on the same search key.
- ▶ In a dense non clustering index, the index must store a list of pointers to all records with the same search-key value.

Dense Indices Example

10101		→	10101	Srinivasan	Comp. Sci.	65000	
12121		→	12121	Wu	Finance	90000	
15151		→	15151	Mozart	Music	40000	
22222		→	22222	Einstein	Physics	95000	
32343		→	32343	El Said	History	60000	
33456		→	33456	Gold	Physics	87000	
45565		→	45565	Katz	Comp. Sci.	75000	
58583		→	58583	Califieri	History	62000	
76543		→	76543	Singh	Finance	80000	
76766		→	76766	Crick	Biology	72000	
83821		→	83821	Brandt	Comp. Sci.	92000	
98345		→	98345	Kim	Elec. Eng.	80000	

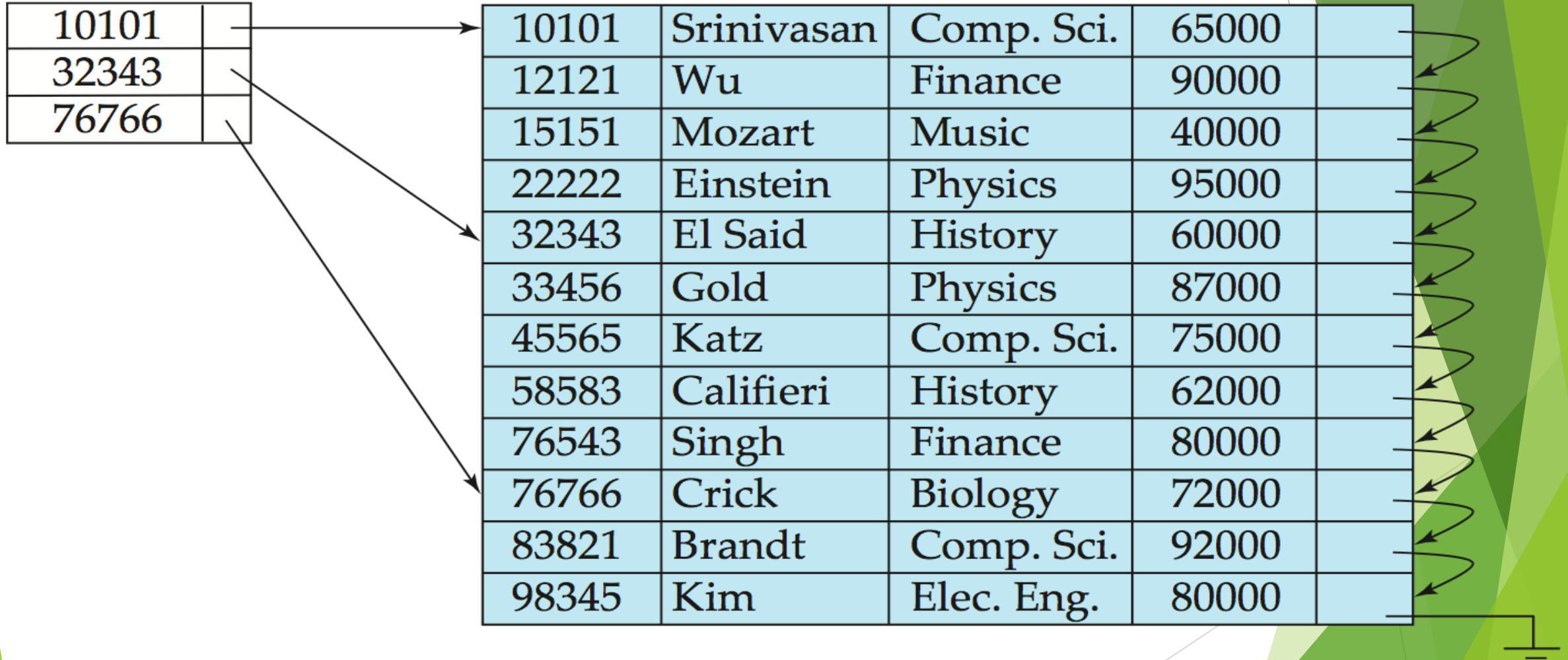
Dense Index Example



Sparse Indices

- ▶ In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index
- ▶ 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.
- ▶ Compared to dense indices:
- ▶ Less space and less maintenance overhead for insertions and deletions.
- ▶ Generally slower than dense index for locating records.

Sparse Index Example



Index Updation

► Insertion.

- First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:
 - ◦ Dense indices:
 - 1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.**
 - 2. Otherwise the following actions are taken:**
 - a. If the index entry stores pointers to all records with the same search key value, the system adds a pointer to the new record in the index entry.
 - b. Otherwise, the index entry 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.

Index Updation

- ▶ 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 (in search-key order) appearing in the new block into the index.
- ▶ On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

Index Updation

Deletion: To delete a record, the system first looks up the record to be deleted.

- ▶ The actions the system takes next depend on whether the index is dense or sparse:
- ▶ ◦ Dense indices:
- ▶ 1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.
- 2. Otherwise the following actions are taken:
 - a. If the index entry stores pointers to all records with the same search key value, the system deletes the pointer to the deleted record from the index entry.
 - b. Otherwise, the index entry 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 entry to point to the next record.

Query Processing

- ▶ Query processing refers to the range of activities involved in extracting data from a database.
- ▶ The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries
- ▶ Basic Steps in Query Processing are
 1. Parsing and translation
 2. Optimization
 3. Evaluation

Query Processing

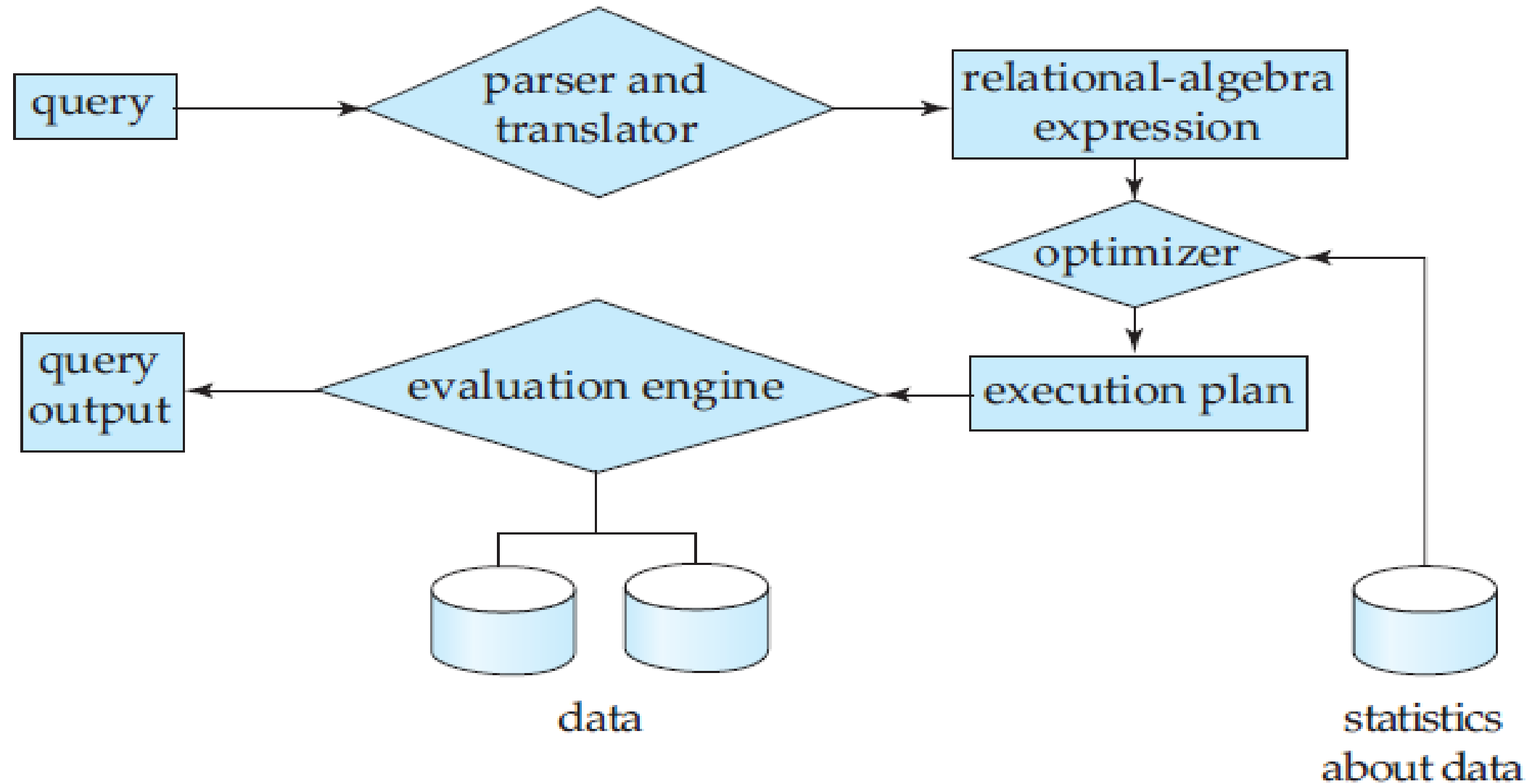


Figure 12.1 Steps in query processing.

Steps in Query Processing

1 Parsing and translation

- ▶ The first action the system must take in query processing is to translate a given query into its internal form.
- ▶ In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on.
- ▶ The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression
- ▶ If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.

Steps in Query Processing

1 Parsing and translation

► consider the query:

select salary
from instructor
where salary < 75000;

This query can be translated into either of the following relational-algebra expressions:

$\sigma_{salary < 75000} (\pi_{salary} (instructor))$

$\pi_{salary} (\sigma_{salary < 75000} (instructor))$

Steps in Query Processing

2 Evaluation

- ▶ A relational algebra operation annotated with instructions on how to evaluate it is called an evaluation primitive.
- ▶ A sequence of primitive operations that can be used to evaluate a query is a query-execution plan or query-evaluation plan
- ▶ The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

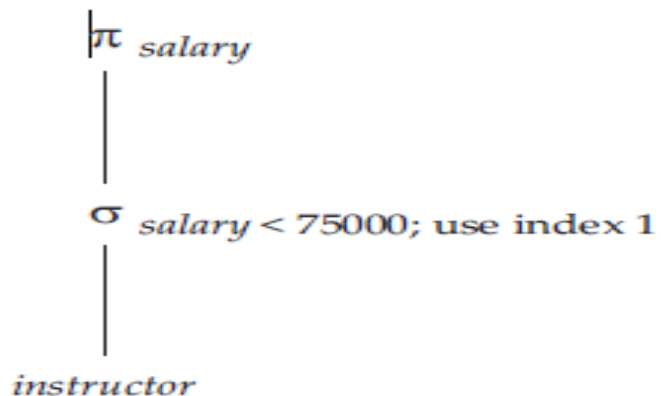


Figure 12.2 A query-evaluation plan.

Steps in Query Processing

2 Evaluation

- ▶ A relational algebra operation annotated with instructions on how to evaluate it is called an evaluation primitive.
- ▶ A sequence of primitive operations that can be used to evaluate a query is a query-execution plan or query-evaluation plan
- ▶ The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

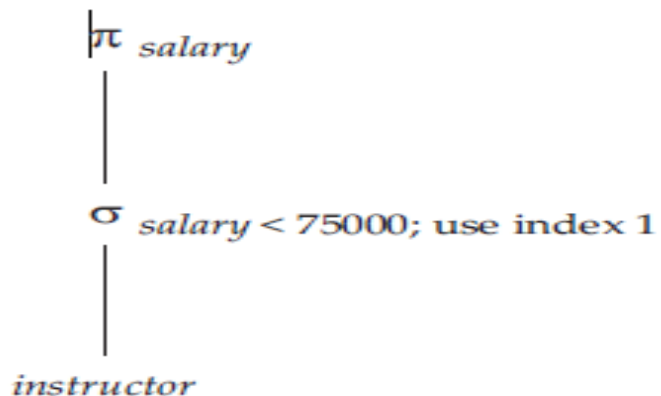


Figure 12.2 A query-evaluation plan.

Steps in Query Processing

3 Query Optimization

- The different evaluation plans for a given query can have different costs
- ▶ Amongst all equivalent evaluation plans choose the one with lowest cost.
- ▶ Cost is estimated using statistical information from the database catalog
 - e.g. number of tuples in each relation, size of tuples, etc.

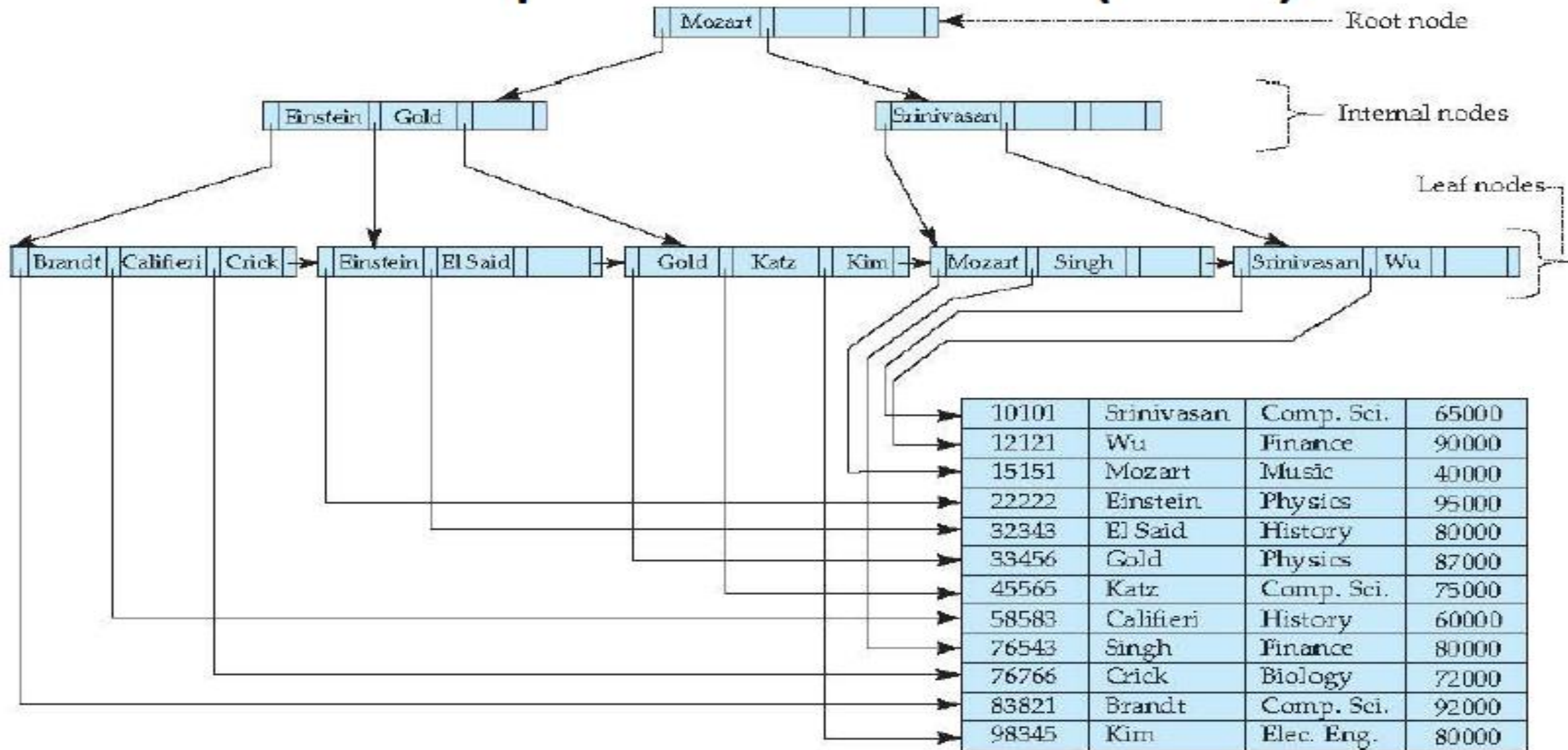
B+ Tree

- ▶ A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.
- ▶ Implementation of dynamic multilevel index structure
- ▶ In a B+ tree, data pointers are stored only at the leaf nodes of the tree, hence, the structure of leaf nodes differs from the structure of internal nodes
- ▶ The leaf nodes have an entry for every value of search field, along with a data pointer to the record if the search field is a key field.
- ▶ For a non key search field, the pointers point to a block containing pointers to the data field records, creating an extra level of indirection

B+tree indices are an alternative to indexed-sequential files

- ▶ Disadvantage of indexed-sequential files
 - ▶ performance degrades as file grows, since many overflow blocks get created.
 - ▶ Periodic reorganization of entire file is required.
- ▶ Advantage of B⁺-tree index files:
 - ▶ automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
 - ▶ Reorganization of entire file is not required to maintain performance.
- ▶ (Minor) disadvantage of B⁺-trees:
 - ▶ extra insertion and deletion overhead, space overhead.
- ▶ Advantages of B⁺-trees outweigh disadvantages
 - ▶ B⁺-trees are used extensively

Example of B⁺-Tree(n=4)



B+ Tree Node Structure

Typical Node



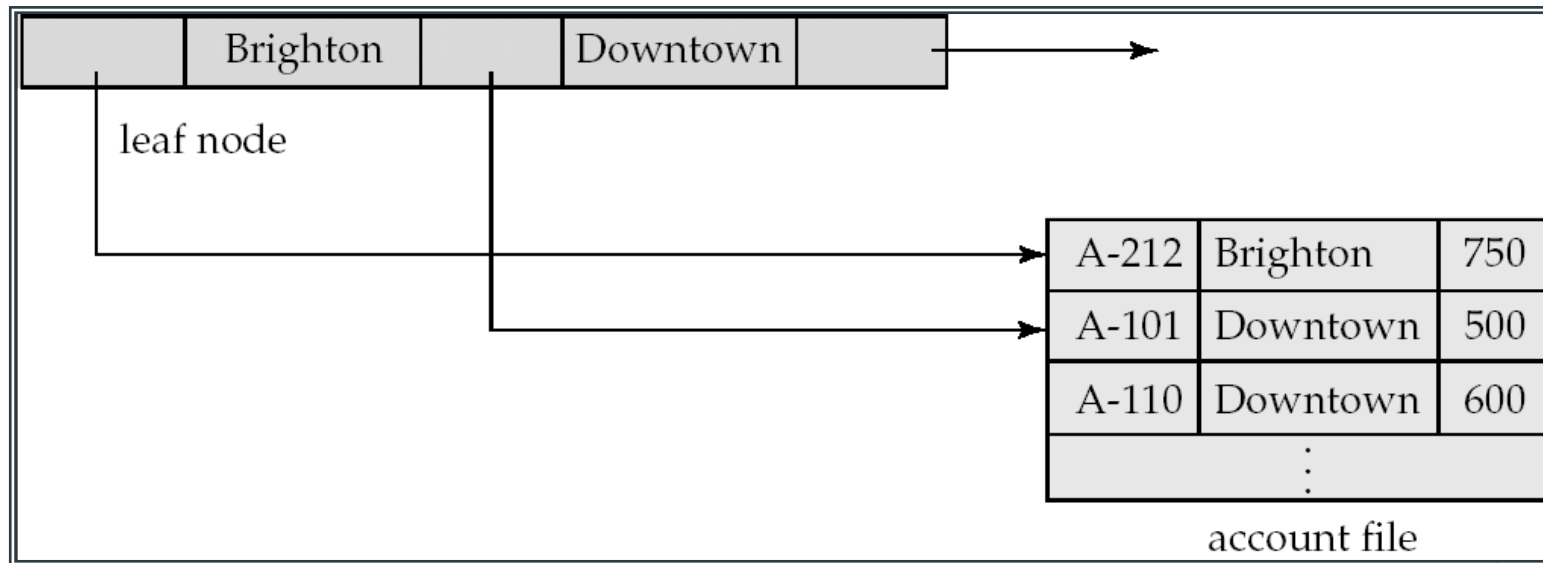
- ▶ K_i are the search-key values
- ▶ P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- ▶ The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

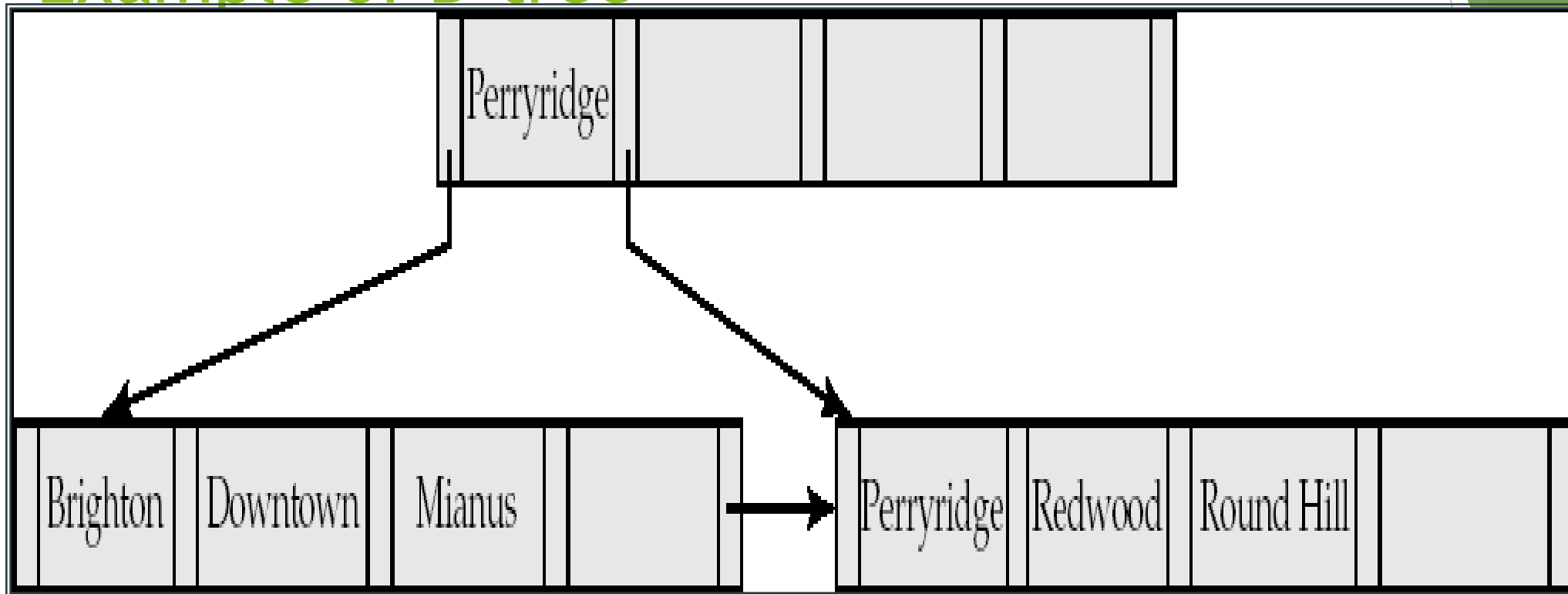
Leaf Node in B+ Tree

Properties of leaf node are

- ▶ For $i = 1, 2, \dots, n-1$, pointer P_i either points to a file record with search-key value K_i , or to a bucket of pointers to file records, each record having search-key value K_i . Only need bucket structure if search-key does not form a primary key.
- ▶ If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than L_j 's search-key values
- ▶ P_n points to next leaf node in search-key order



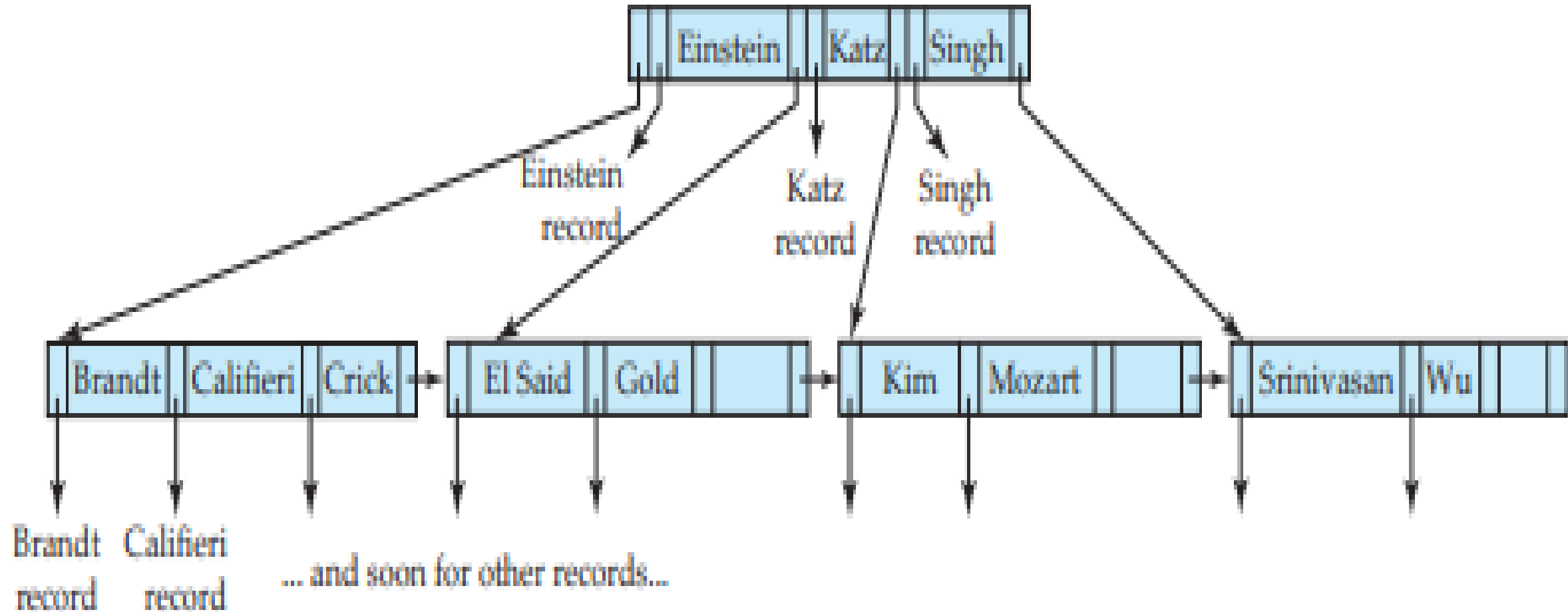
Example of B⁺tree



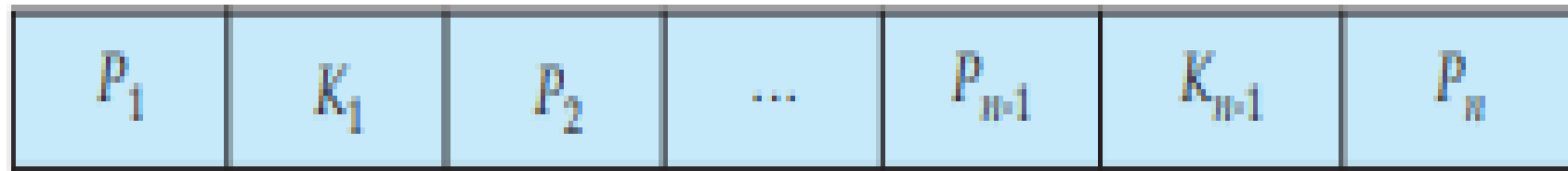
B tree Index

- ▶ B-tree indices are similar to B+-tree indices. 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 (if they are unique), unlike a B+-tree, where a value may appear in a non leaf node, in addition to appearing in a leaf node
- ▶ Since search keys are not repeated in the B-tree, we may be able to store the index in fewer tree nodes than in the corresponding B+-tree index
- ▶ Since search keys that appear in non leaf nodes appear nowhere else in the B-tree, we are forced to include an additional pointer field for each search key in a non leaf node. These additional pointers point to either file records or buckets for the associated search key

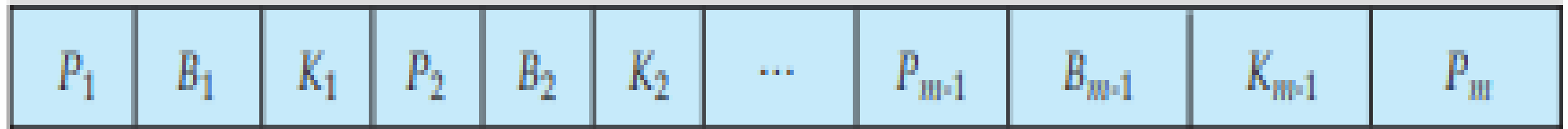
B tree Index



B tree Index



(a)



(b)

Figure 11.22 Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

B tree Index

- ▶ Pointer P_i either points to a file record with search-key value K_i , or to a bucket of pointers to file records, each record having search-key value K_i .
- ▶ Pointers B_i are bucket or file-record pointers. In the generalized B-tree in the figure, there are $n-1$ keys in the leaf node, but there are $m-1$ keys in the non leaf node
- ▶ The number of nodes accessed in a lookup in a B-tree depends on where the search key is located.

Selection Operation

- ▶ In query processing, the file scan is the lowest-level operator to access data.
- ▶ **File** scans are search algorithms that locate and retrieve records that fulfill a selection condition
- ▶ In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file

A1 (linear search).

- In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition.
- An initial seek is required to access the first block of the file.
- In case blocks of the file are not stored contiguously, extra seeks may be required
- ▶ Cost is $tS + br * tT$ (One initial seek plus br block transfers, where br denotes the number of blocks in the file.)

Selection Operation

- ▶ Index structures are referred to as **access paths**, since they provide a **path** through which data can be located and accessed.
- ▶ Primary index (also referred to as a clustering index) is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file.
- ▶ An index that is not a primary index is called a secondary index.
- ▶ Search algorithms that use an index are referred to as **index scans**
- ▶ Search algorithms that use an index are:
 - ❑ **A2 (primary index, equality on key).**
 - ❑ **A3 (primary index, equality on nonkey)**
 - ❑ **A4 (secondary index, equality).**

Selection Operation

- ▶ Index structures are referred to as **access paths**, since they provide a **path** through which data can be located and accessed.
- ▶ Primary index (also referred to as a clustering index) is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file.
- ▶ An index that is not a primary index is called a secondary index.
- ▶ Search algorithms that use an index are referred to as **index scans**
- ▶ Search algorithms that use an index are:
 - ❑ **A2 (primary index, equality on key).**
 - ❑ **A3 (primary index, equality on nonkey)**
 - ❑ **A4 (secondary index, equality).**

Selection Operation

A2 (primary index, equality on key).

- ▶ For an equality comparison on a key attribute with a primary index, we can use the index to retrieve a single record that satisfies the corresponding equality condition
- ▶ $Cost = (h_i + 1) * (t_T + t_S)$

A3 (primary index, equality on nonkey).

- We can retrieve multiple records by using a primary index when the selection condition specifies an equality comparison on a nonkey attribute, A.
- The only difference from the A2 is that multiple records may need to be fetched. However, the records must be stored consecutively in the file since the file is sorted on the search key
- Let b = number of blocks containing matching records
 - ▶ $Cost = h_i * (t_T + t_S) + t_T * b$

Selection Operation

- ▶ **A4 (secondary index, equality).**
- ▶ **Selections specifying an equality condition** can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may be retrieved if the indexing field is not a key.
- ▶ In the first case, only one record is retrieved. The time cost in this case is the same as that for a primary index (caseA2).
- ▶ In the second case, each record may be resident on a different block, which may result in one I/O operation per retrieved record, with each I/O operation requiring a seek and a block transfer.
- ▶ The worst-case time cost in this case is $(h_i + n) * (t_S + t_T)$, where n is the number of records fetched, if each record is in a different disk block, and the block fetches are randomly ordered

Selection Operation

A5 (**primary index, comparison**). (Relation is sorted on A)

- ▶ For comparisons of the form $A < v$ or $A \leq v$, an index lookup is not required.
- ▶ For $A < v$, we use a simple file scan starting from the beginning of the file, and continuing up to (but not including) the first tuple with attribute $A = v$.
- ▶ The case of $A \leq v$ is similar, except that the scan continues up to (but not including) the first tuple with attribute $A > v$.

A6 (**secondary index, comparison**).

- We can use a secondary ordered index to guide retrieval for comparison conditions involving $<, \leq, \geq$, or $>$.
- The lowest-level index blocks are scanned, either from the smallest value up to v (for $<$ and \leq), or from v up to the maximum value (for $>$ and \geq).
- ▶ The secondary index provides pointers to the records, but to get the actual records we have to fetch the records by using the pointers

Selection Operation

Implementation of Complex Selections

► complex selection predicates are

❖ **Conjunction:** A *conjunctive selection* is a selection of the form:

$$\sigma_{\phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_n}(r)$$

❖ **Disjunction:** A *disjunctive selection* is a selection of the form:

$$\sigma_{\phi_1 \vee \phi_2 \vee \dots \vee \phi_n}(r)$$

► A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions ϕ_i

❖ **Negation:** The result of a selection $\sigma_{\neg \phi}(r)$ is the set of tuples of r for which the condition evaluates to false. In the absence of nulls, this set is simply the set of tuples in r that are not in $\sigma_{\phi}(r)$.

Selection Operation

- ▶ Selection operation involving either a conjunction or a disjunction of simple conditions can be implemented using following Algorithms
 - **A7 (conjunctive selection using one index).**
- ▶ In this, first determine whether an access path is available for an attribute in one of the simple conditions.
- ▶ If one is, one of the selection algorithms A2 through A6 can retrieve records satisfying that condition.
- ▶ Complete the operation by testing, in the memory buffer, whether or not each retrieved record satisfies the remaining simple conditions.
- ▶ To reduce the cost, we choose a \emptyset_i and one of algorithms A1 through A6 for which the combination results in the least cost for $\sigma_{\emptyset_i}(r)$

Selection Operation

A8 (conjunctive selection using composite index).

- An appropriate *composite index* (that is, an index on multiple attributes) may be available for some conjunctive selections.
- If the selection specifies an equality condition on two or more attributes, and a composite index exists on these combined attribute fields, then the index can be searched directly.
- The type of index determines which of algorithms A2, A3, or A4 will be used

Selection Operation

A9 (conjunctive selection by intersection of identifiers).

- **Another alternative** for implementing conjunctive selection operations involves the use of record pointers or record identifiers.
- This algorithm requires indices with record pointers, on the fields involved in the individual conditions.
- The algorithm scans each index for pointers to tuples that satisfy an individual condition
- The intersection of all the retrieved pointers is the set of pointers to tuples that satisfy the conjunctive condition.
- The algorithm then uses the pointers to retrieve the actual records.
- If indices are not available on all the individual conditions, then the algorithm tests the retrieved records against the remaining conditions.

Selection Operation

A9 (conjunctive selection by intersection of identifiers).

- ▶ The cost of algorithm A9 is the sum of the costs of the individual index scans, plus the cost of retrieving the records in the intersection of the retrieved lists of pointers.
- ▶ This cost can be reduced by sorting the list of pointers and retrieving records in the sorted order.
- ▶ Thereby, (1) all pointers to records in a block come together, hence all selected records in the block can be retrieved using a single I/O operation, and (2) blocks are read in sorted order, minimizing disk-arm movement

Selection Operation

A10 (disjunctive selection by union of identifiers).

- ▶ If access paths are available on all the conditions of a disjunctive selection, each index is scanned for pointers to tuples that satisfy the individual condition.
- ▶ The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy the disjunctive condition. We then use the pointers to retrieve the actual records
- ▶ However, if even one of the conditions does not have an access path, we have to perform a linear scan of the relation to find tuples that satisfy the condition.
- ▶ Therefore, if there is even one such condition in the disjunct, the most efficient access method is a linear scan, with the disjunctive condition tested on each tuple during the scan.

	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies condition, scan can be terminated as soon as the required record is found. In the worst case, b_r blocks transfers are still required.
A2	Primary B ⁺ -tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3	Primary B ⁺ -tree Index, Equality on Nonkey	$h_i * (t_T + t_S) + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a primary index) and don't require additional seeks.

A4	Secondary B ⁺ -tree Index, Equality on Nonkey	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Primary B ⁺ -tree Index, Comparison	$h_i * (t_T + t_S) + b * t_T$	Identical to the case of A3, equality on nonkey.
A6	Secondary B ⁺ -tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on nonkey.

Figure 12.3 Cost estimates for selection algorithms.

Static Hashing

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

Static Hashing

- ▶ An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- ▶ Ideal hash function is random, so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file.
- ▶ Typical hash functions perform computation 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.

Static Hashing

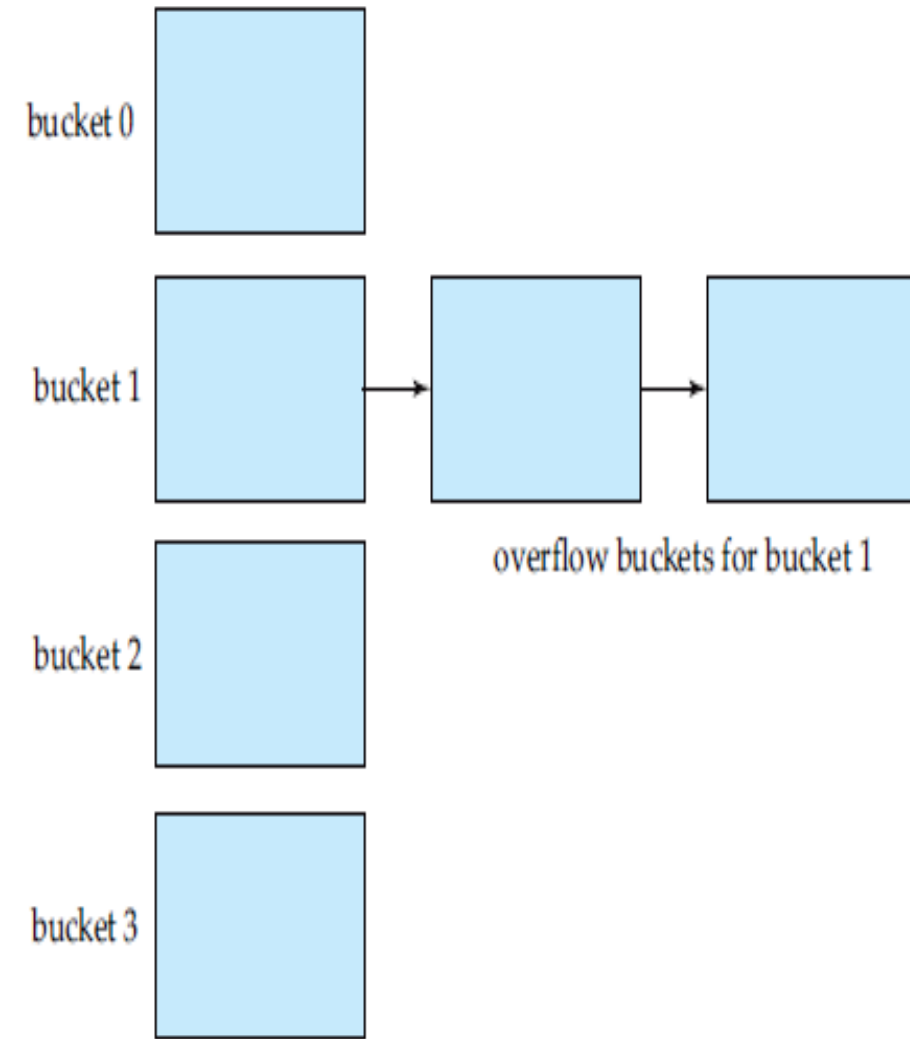
Handling of Bucket Overflows

- ▶ Bucket overflow can occur because of
 - ▶ Insufficient buckets
 - ▶ Skew in distribution of records. This can occur due to two reasons:
 - ▶ Multiple records have same search-key value
 - ▶ Chosen hash function produces non-uniform distribution of key values
 - ▶ Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using overflow buckets.

Static Hashing

Bucket Overflow

- ▶ The condition of bucket-overflow is known as **collision**.
- ▶ Overflow Chaining – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called Closed Hashing
- ▶ Linear Probing – When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called Open Hashing.



Overflow chaining in a hash structure.

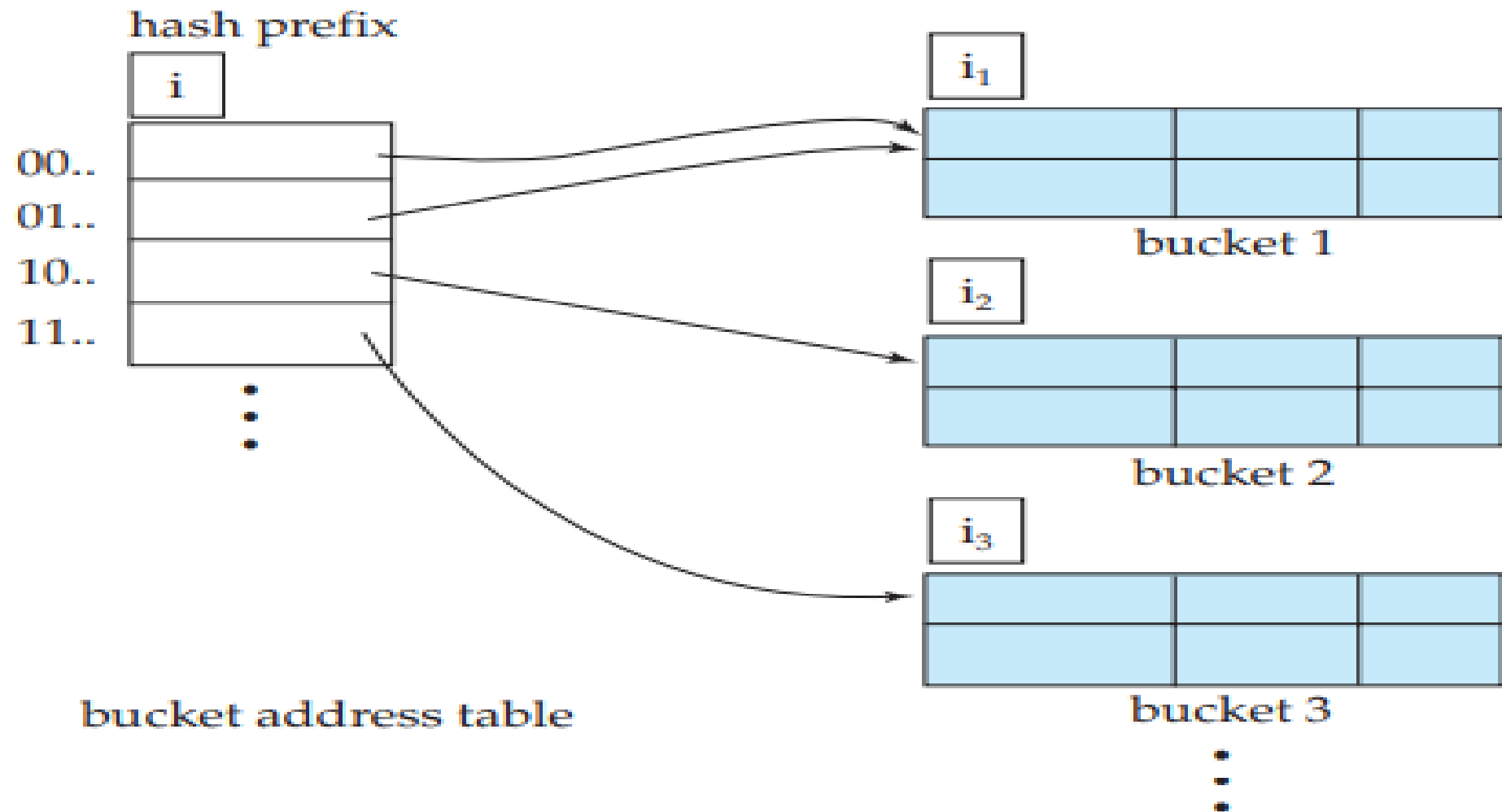
Deficiencies of static hashing

- ▶ In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
- ▶ If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
- ▶ If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
- ▶ If database shrinks, again space will be wasted.
- ▶ One solution: periodic re-organization of the file with a new hash function – Expensive, disrupts normal operations
- ▶ Better solution: 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
- ▶ Hash function generates values over a large range — typically b -bit integers, with $b = 32$.
- ▶ At any time use only a prefix of the hash function to index into a table of bucket addresses. Let the length of the prefix be i bits, $0 \leq i \leq 32$. • Bucket address table size = 2^i .
- ▶ Initially $i = 0$ • Value of i grows and shrinks as the size of the database grows and shrinks. – Multiple entries in the bucket address table may point to a bucket (why?) – Thus, actual number of buckets is $< 2^i$ • The number of buckets also changes dynamically due to coalescing and splitting of buckets.

Dynamic Hashing



13-10-2021

Figure 11.26 General extendable hash structure.