Database Management Systems (CSN-351)
Indexing and Hashing

BTech 3rd Year (CS) + Minor + Audit

Instructor: **R**anita **B**iswas
Department of Computer Science and Engineering
Indian Institute of Technology Roorkee
Roorkee, Uttarakhand - 247 667, India

Indexing

Indexing mechanisms used to speed up access to desired data.
E.g., author catalog in library

Indexing

Indexing mechanisms used to speed up access to desired data.
E.g., author catalog in library

Search Key: attribute or set of attributes which is used to look up records in a file.

Indexing

Indexing mechanisms used to speed up access to desired data.
E.g., author catalog in library

Search Key: attribute or set of attributes which is used to look up records in a file.

An index file consists of records (called index entries) of the form

search-key	pointer

Indexing

Indexing mechanisms used to speed up access to desired data.
E.g., author catalog in library

Search Key: attribute or set of attributes which is 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.

Indexing

Indexing mechanisms used to speed up access to desired data.
E.g., author catalog in library

Search Key: attribute or set of attributes which is 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.

Ordered indices: Based on a sorted ordering of the values.

Indexing

Indexing mechanisms used to speed up access to desired data.
E.g., author catalog in library

Search Key: attribute or set of attributes which is 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.

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*.

Indexing Techniques Evaluation Factors

- Access types
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.

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**.

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.

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**.

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.

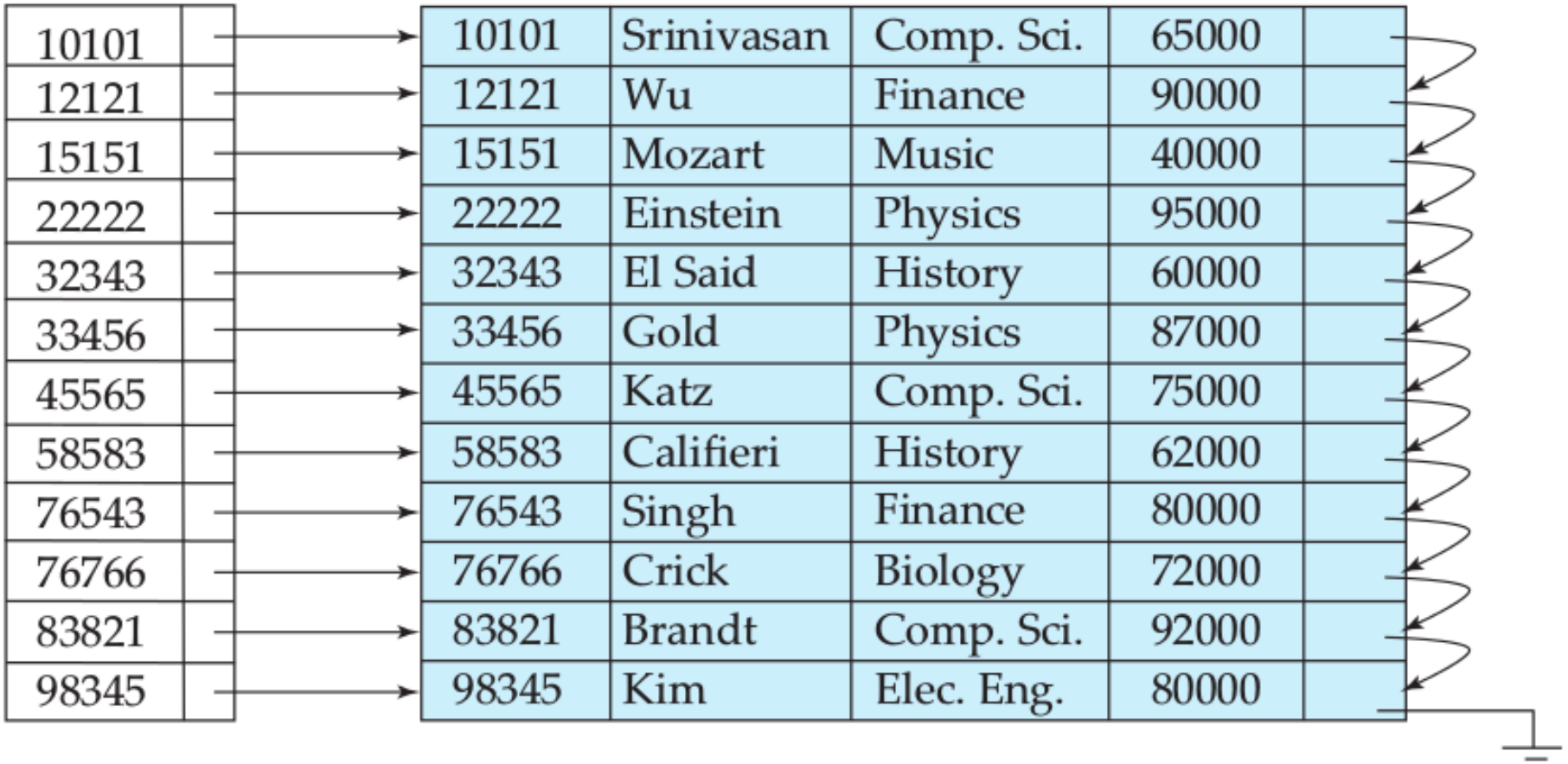
Secondary index: an index whose search key specifies an order different from the sequential order of the file.

Also called **non-clustering index**.

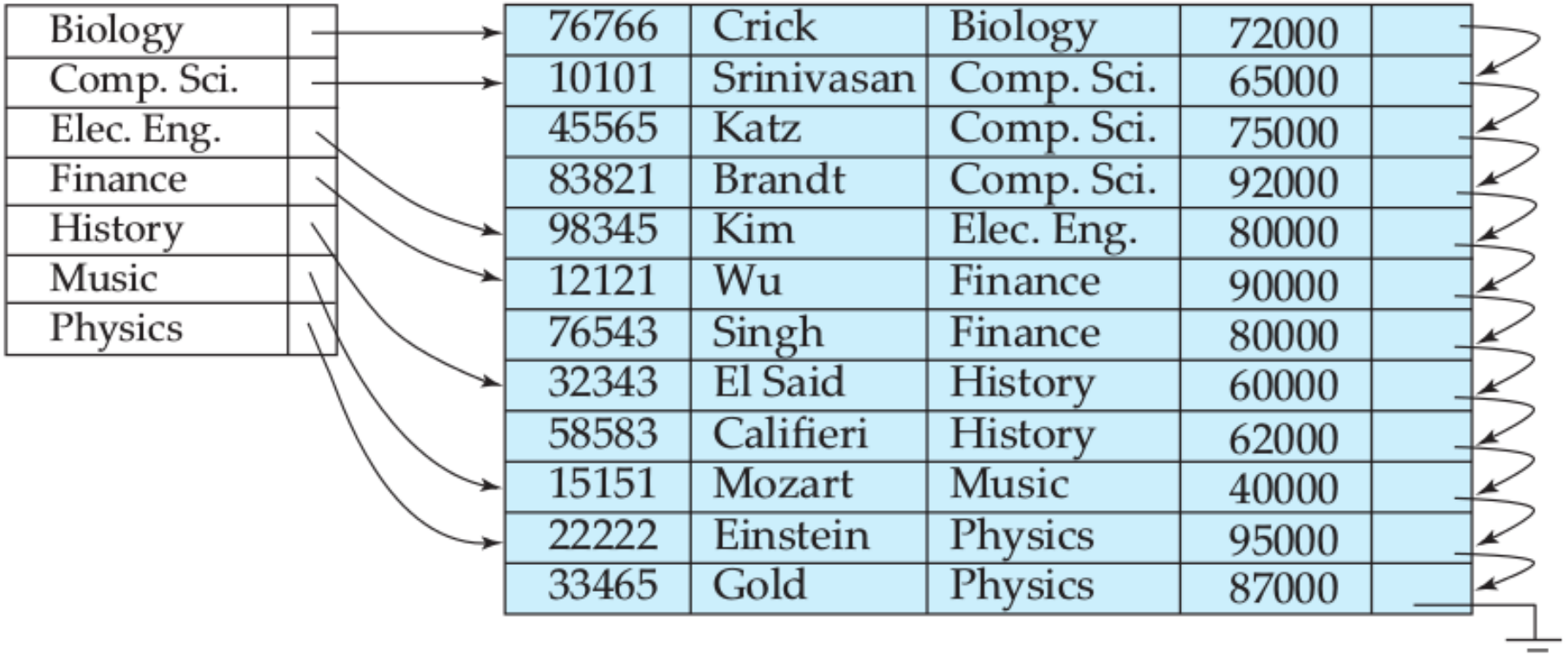
Index-sequential file: ordered sequential file with a primary index.

Dense Index Files

Dense index: Index record appears for every search-key value in the file.



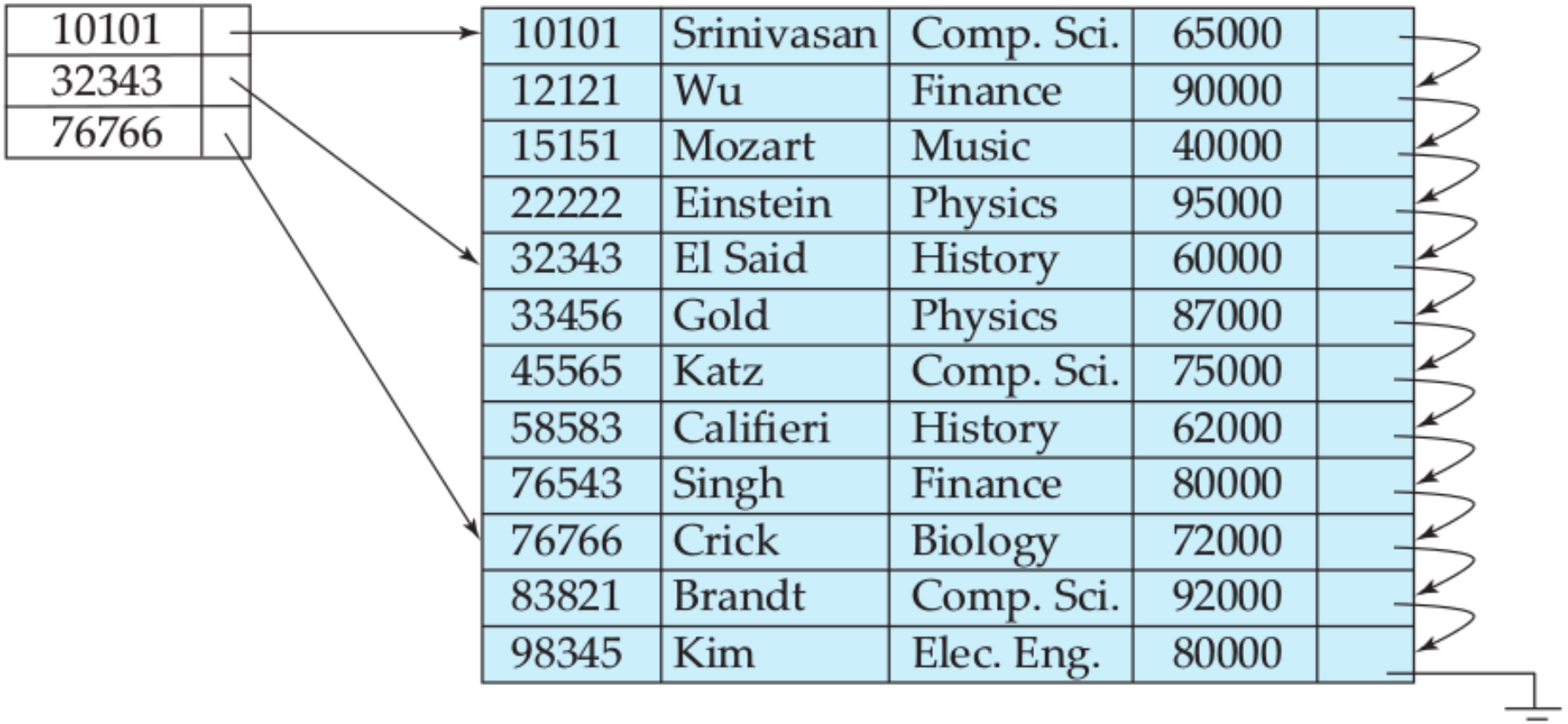
Dense Index Files



Sparse Index Files

Sparse Index: contains index records for only some search-key values.

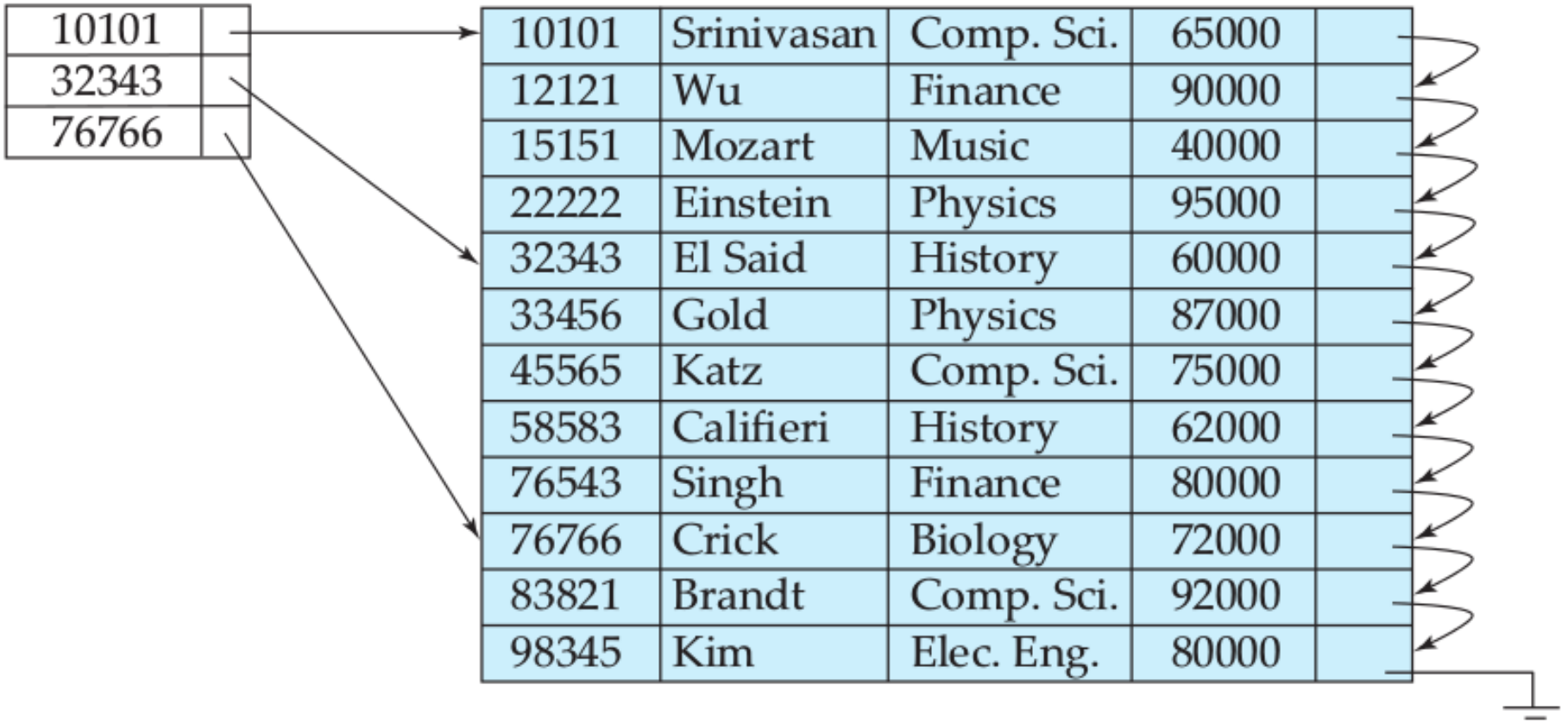
Applicable when records are sequentially ordered on search-key.



Sparse Index Files

To locate a record with search-key value `K` we:

- Find index record with largest search-key value $< K$.
- Search file sequentially starting at the record to which the index record points.



Sparse Index Files

Compared to dense indices:

Sparse Index Files

Compared to dense indices:

- Less space and less maintenance overhead for insertions and deletions.

Sparse Index Files

Compared to dense indices:

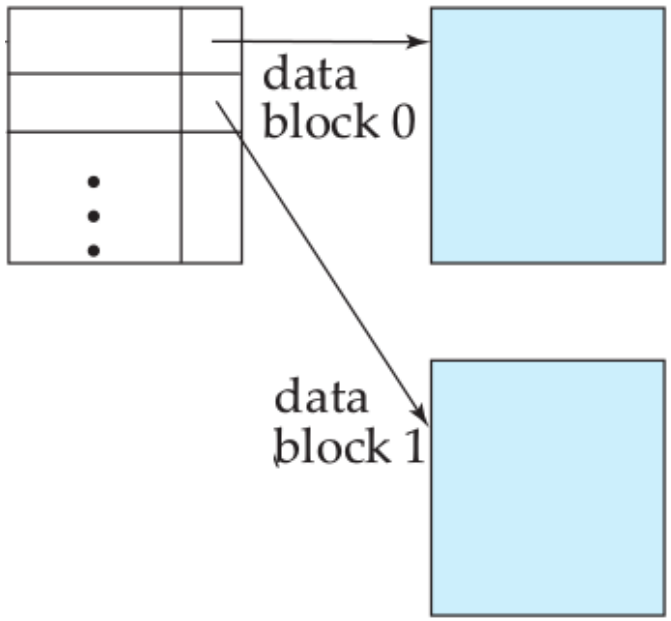
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.

Sparse Index Files

Compared to dense indices:

- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.

Good tradeoff: sparse index with an index entry for every block in file, corresponding to the smallest search-key value in the block.



Multilevel Indices

If primary index does not fit in memory, access becomes expensive.

Multilevel Indices

If primary index does not fit in memory, access becomes expensive.

Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.

- **outer index:** a sparse index of primary index
- **inner index:** the primary index file

Multilevel Indices

If primary index does not fit in memory, access becomes expensive.

Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.

- **outer index:** a sparse index of primary index
- **inner index:** the primary index file

If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.

Multilevel Indices

If primary index does not fit in memory, access becomes expensive.

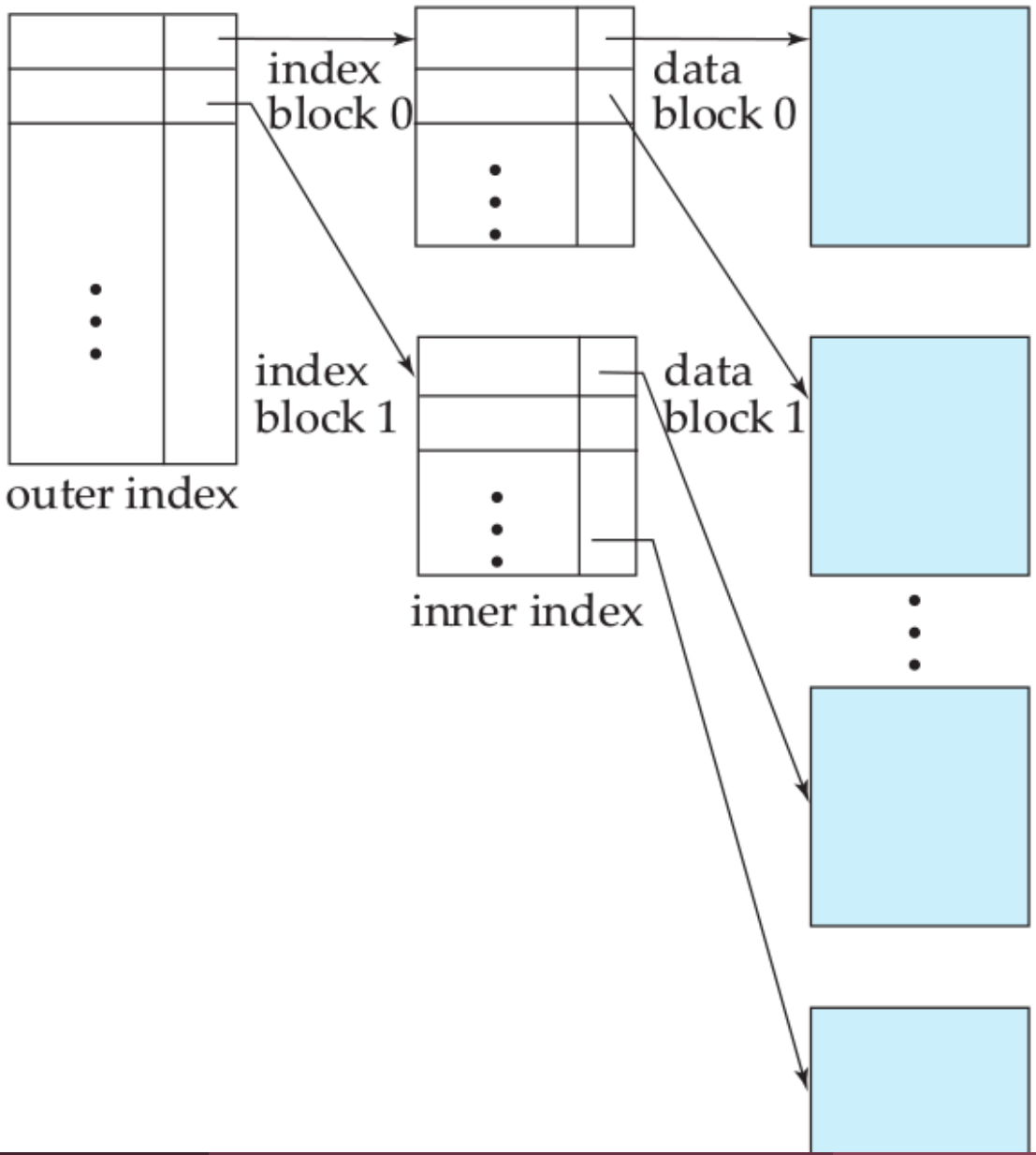
Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.

- **outer index:** a sparse index of primary index
- **inner index:** the primary index file

If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.

Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Indices



Index Update: Deletion

If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

Index Update: Deletion

If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

Single-level index entry deletion:

- **Dense indices:**
Deletion of search-key is similar to file record deletion.

Index Update: Deletion

If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

Single-level index entry deletion:

- **Dense indices:**
Deletion of search-key is similar to file record deletion.

- **Sparse indices:**
If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).

Index Update: Deletion

If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

Single-level index entry deletion:

- **Dense indices:**
Deletion of search-key is similar to file record deletion.

- **Sparse indices:**
If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).

If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

Single-level index insertion:

- Perform a lookup using the search-key value appearing in the record to be inserted.

Index Update: Insertion

Single-level index insertion:

- Perform a lookup using the search-key value appearing in the record to be inserted.
- **Dense indices:**

If the search-key value does not appear in the index, insert it.

Index Update: Insertion

Single-level index insertion:

- Perform a lookup using the search-key value appearing in the record to be inserted.
- **Dense indices:**
If the search-key value does not appear in the index, insert it.
- **Sparse indices:**
If index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.

Index Update: Insertion

Single-level index insertion:

- Perform a lookup using the search-key value appearing in the record to be inserted.
- **Dense indices:**
If the search-key value does not appear in the index, insert it.
- **Sparse indices:**
If index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.

If a new block is created, the first search-key value appearing in the new block is inserted into the index.

Index Update: Insertion

Single-level index insertion:

- Perform a lookup using the search-key value appearing in the record to be inserted.
- **Dense indices:**
If the search-key value does not appear in the index, insert it.
- **Sparse indices:**
If index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.

If a new block is created, the first search-key value appearing in the new block is inserted into the index.

Multilevel insertion and deletion: algorithms are simple extensions of the single-level algorithms.

Secondary Indices

Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.

Secondary Indices

Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.

Example 1: In the `instructor` relation stored sequentially by `ID`, we may want to find all instructors in a particular department.

Secondary Indices

Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.

Example 1: In the `instructor` relation stored sequentially by `ID`, we may want to find all instructors in a particular department.

Example 2: We want to find all instructors with a specified salary or with salary in a specified range of values.

Secondary Indices

Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.

Example 1: In the `instructor` relation stored sequentially by `ID`, we may want to find all instructors in a particular department.

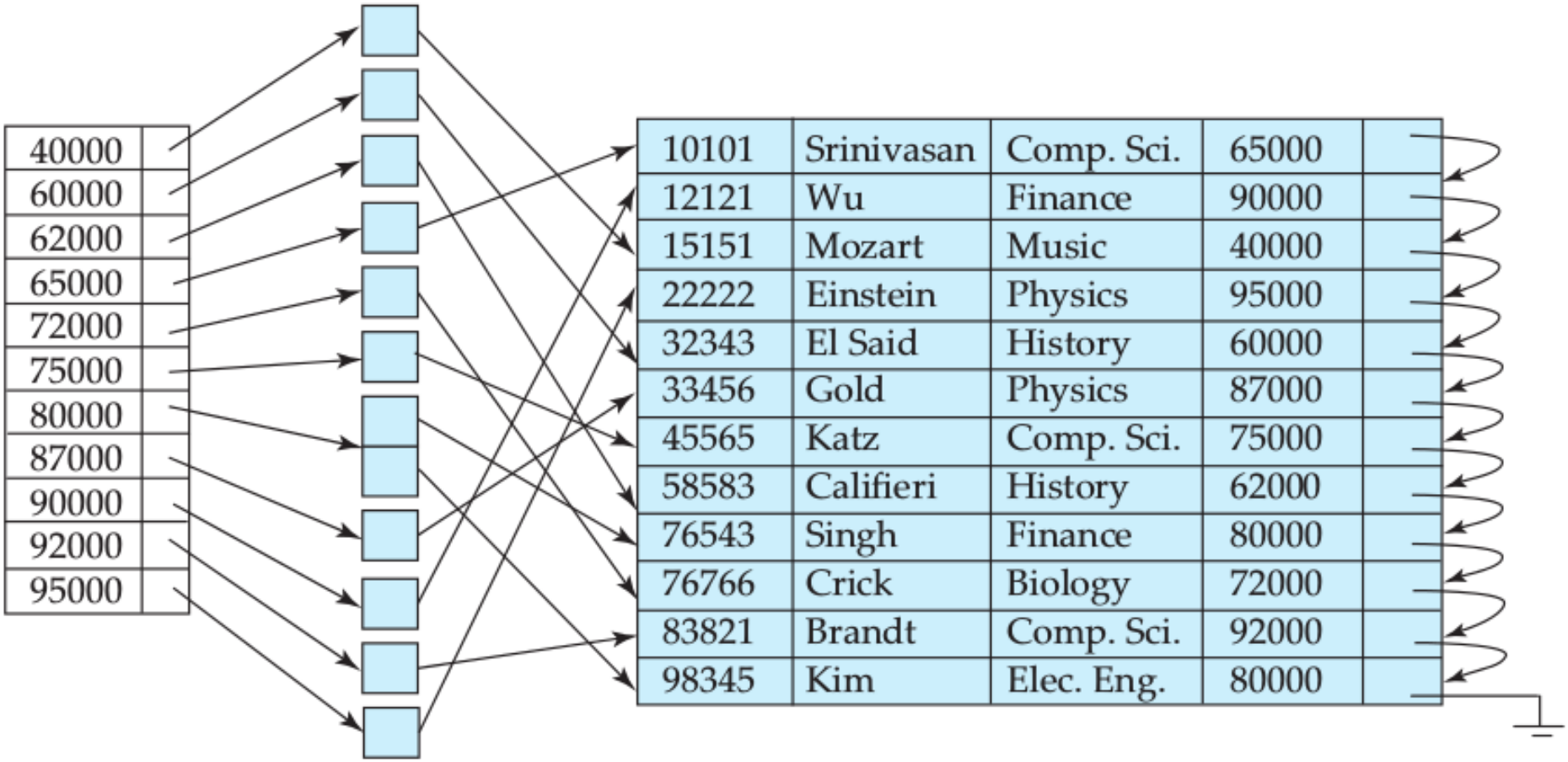
Example 2: We want to find all instructors with a specified salary or with salary in a specified range of values.

We can have a **secondary index** with an index record for each search-key value.

Secondary Indices

Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

Secondary indices have to be dense



Primary and Secondary Indices

Indices offer substantial benefits when searching for records.

Primary and Secondary Indices

Indices offer substantial benefits when searching for records.

BUT: Updating indices imposes overhead on database modification — when a file is modified, every index on the file must be updated.

Primary and Secondary Indices

Indices offer substantial benefits when searching for records.

BUT: Updating indices imposes overhead on database modification — when a file is modified, every index on the file must be updated.

Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive.

- Each record access may fetch a new block from disk.
- Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access.

Resource

Data Structures Visualization Gallery,
Department of Computer Science, University of San Francisco

`https://www.cs.usfca.edu/~galles/visualization/Algorithms.html`

Question 1

A hash table of length 10 uses open addressing with hash function $h(k) = k \mod 10$, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

- 46, 42, 34, 52, 23, 33
- 34, 42, 23, 52, 33, 46
- 46, 34, 42, 23, 52, 33
- 42, 46, 33, 23, 34, 52

Question 1

A hash table of length 10 uses open addressing with hash function $h(k) = k \mod 10$, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

- ○ 46, 42, 34, 52, 23, 33
- ○ 34, 42, 23, 52, 33, 46
- ● **46, 34, 42, 23, 52, 33**
- ○ 42, 46, 33, 23, 34, 52

Question 2

A hash table of length 10 uses open addressing with hash function $h(k) = k \mod 10$, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

How many different insertion sequences of the key values using the same hash function and linear probing will result in the hash table shown above?

Question 2

A hash table of length 10 uses open addressing with hash function $h(k) = k \mod 10$, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

How many different insertion sequences of the key values using the same hash function and linear probing will result in the hash table shown above?

Answer: **30**

Question 3

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function $h(k) = k \mod 10$ and linear probing. What is the resultant hash table?

Question 4

Consider a hash table of size seven, with starting index zero, and a hash function $(3x + 4)$ mod 7. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing?

Question 5

Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function $x \mod 10$, which of the following statements are true?

i. 9679, 1989, 4199 hash to the same value

ii. 1471, 6171 hash to the same value

iii. All elements hash to the same value

iv. Each element hashes to a different value

Question 5

Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function $x \mod 10$, which of the following statements are true?

i. 9679, 1989, 4199 hash to the same value

ii. 1471, 6171 has to the same value

iii. All elements hash to the same value

iv. Each element hashes to a different value

Question 6

Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions?

- $(97 \times 97 \times 97)/100^3$
- $(99 \times 98 \times 97)/100^3$
- $(97 \times 96 \times 95)/100^3$
- $(97 \times 96 \times 95)/(3! \times 100^3)$

Question 6

Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions?

- $(97 \times 97 \times 97)/100^3$
- $(99 \times 98 \times 97)/100^3$
- $(97 \times 96 \times 95)/100^3$
- $(97 \times 96 \times 95)/(3! \times 100^3)$

Question 7

Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?

- $h(i) = i^2 \mod 10$
- $h(i) = i^3 \mod 10$
- $h(i) = (11 \times i^2) \mod 10$
- $h(i) = (12 \times i) \mod 10$

Question 7

Which one of the following hash functions on integers will distribute keys most uniformly over 10 buckets numbered 0 to 9 for i ranging from 0 to 2020?

- $h(i) = i^2 \mod 10$
- $h(i) = i^3 \mod 10$
- $h(i) = (11 \times i^2) \mod 10$
- $h(i) = (12 \times i) \mod 10$

Question 8

Consider a hash function that distributes keys uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5.

Question 8

Consider a hash function that distributes keys uniformly. The hash table size is 20. After hashing of how many keys will the probability that any new key hashed collides with an existing one exceed 0.5.

Answer: **11**