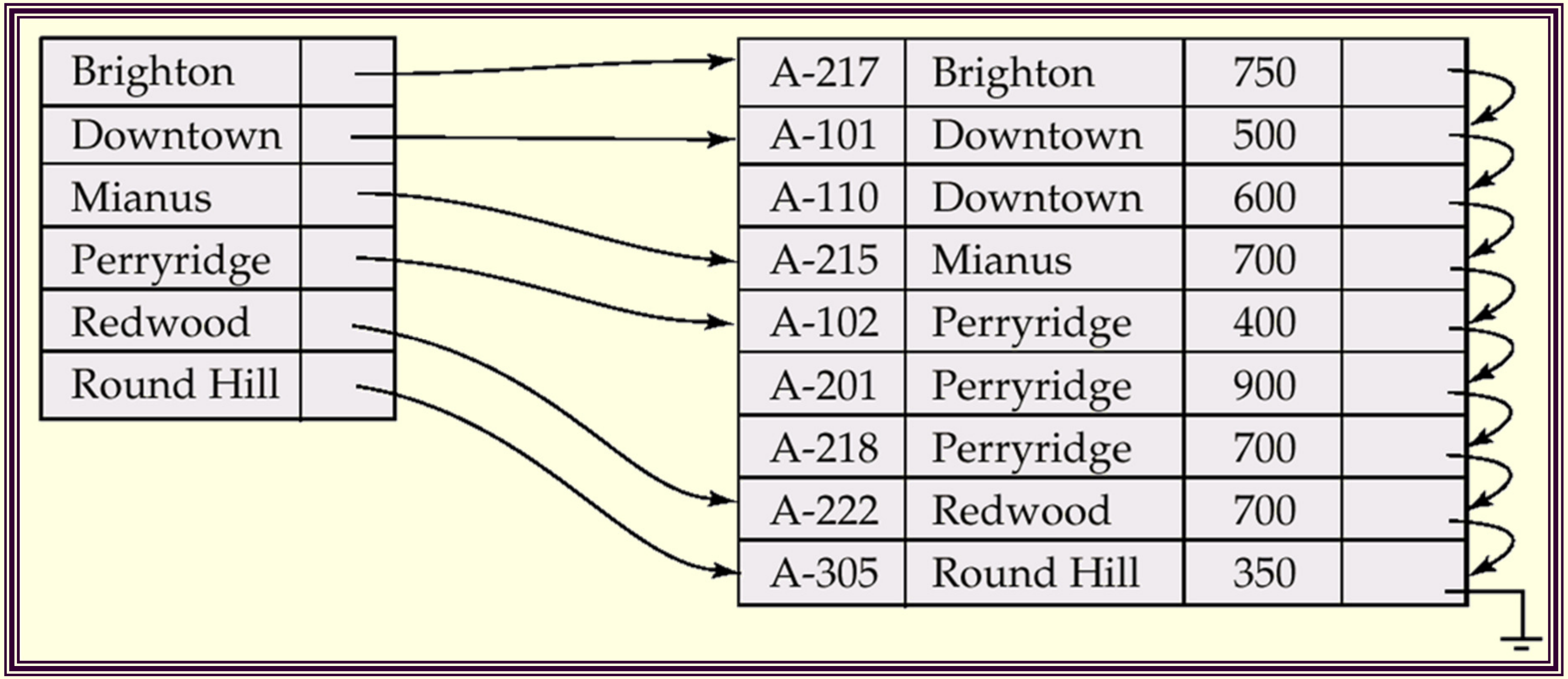
# Dense Index Files

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



# Sparse Index Files

- Sparse index — Index record appears for only some search-key value in the file.

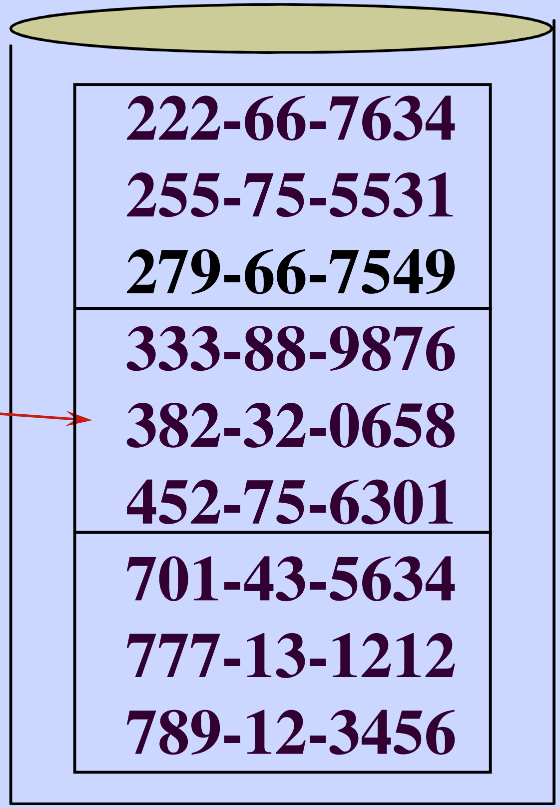
| | |
|---|---|
| Brighton | |
| Mianus | |
| Redwood | |

| | | | |
|---|---|---|---|
| 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 | |

# Indexed Sequential Access: Fast

**Find record with key 382-32-0658**

| Key | Cyl. | Trck | Sect. |
|---|---|---|---|
| 222-66-7634 | 3 | 10 | 2 |
| 333-88-9876 | 3 | 10 | 3 |
| 701-43-5634 | 3 | 10 | 4 |



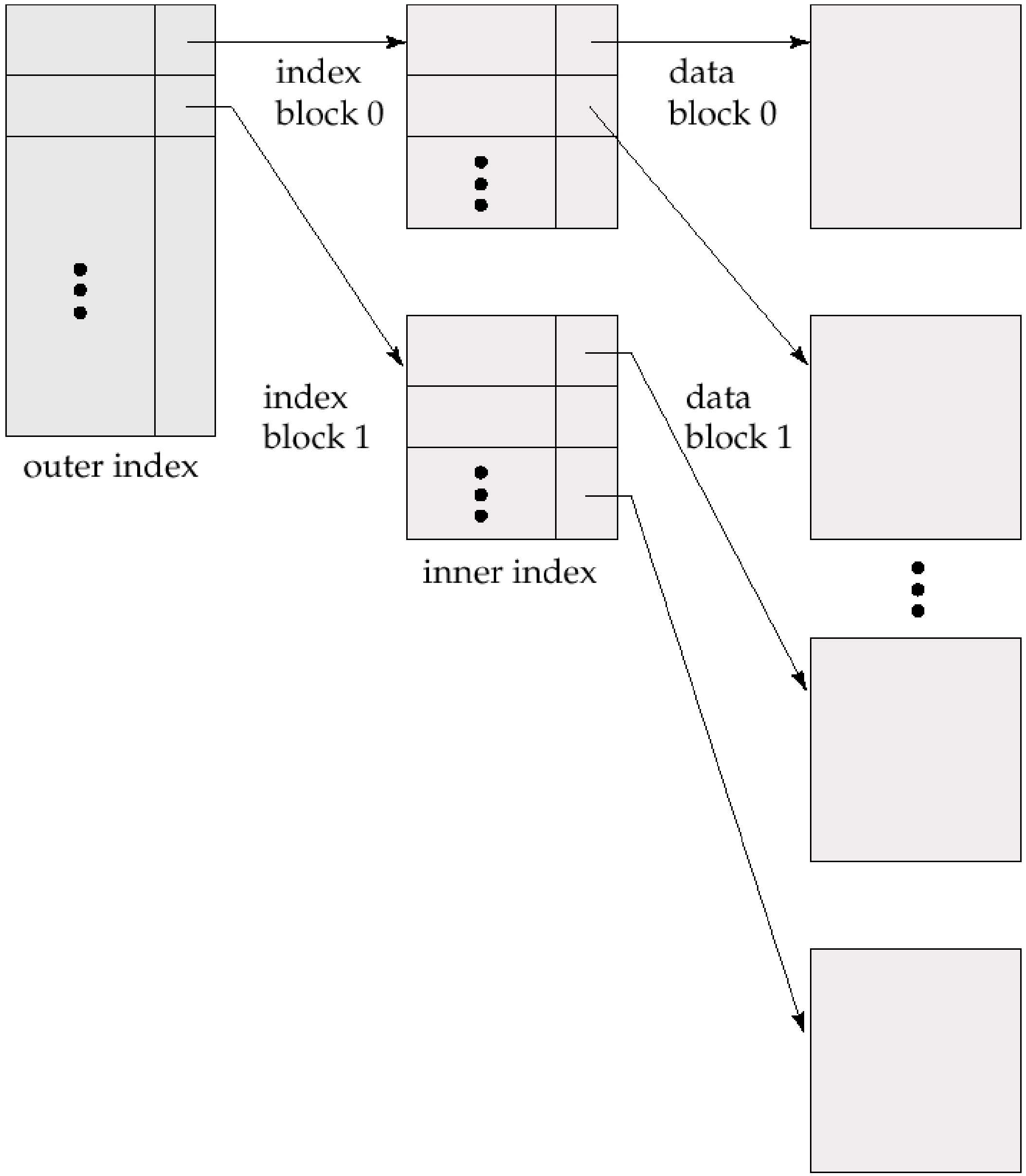

**333-88-9876 < 382-32-0658 > 701-43-5634**

**Search Cyl. 3, Trck 10, Sect. 3 sequentially.**

# Multilevel Index

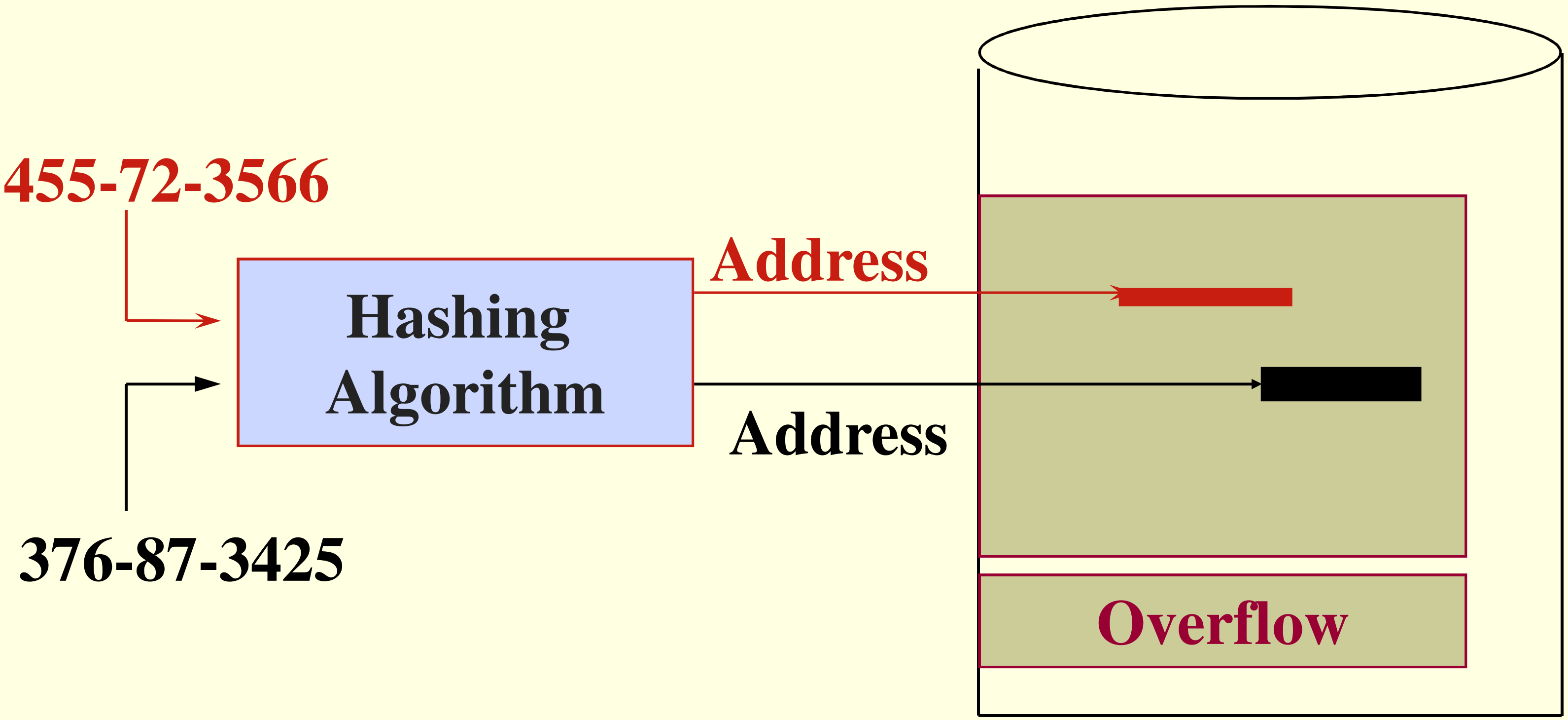
- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.

# Multilevel Index (Cont.)



# Direct or Hashed Access

- A portion of disk space is reserved
- A "hashing" algorithm computes record address



# 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.

# E.g. Hash File Organization

A 1
B 2
C 3
D 4
E 5
G 7
H 8
I 9
L 12
N 14
O 15
P 16
R 18
T 20
U 21
Y 25

- 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(Perryridge) = MOD(16+5+18+18+25+18+9+4+7+5)
= MOD(125) = 5

h(Round Hill) = MOD(18+15+21+14+4+8+9+12+12)
= MOD(113) = 3

h(Brighton) = MOD(2+18+9+7+8+20+15+14)
= MOD(93) = 3

# 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

# Hash Functions

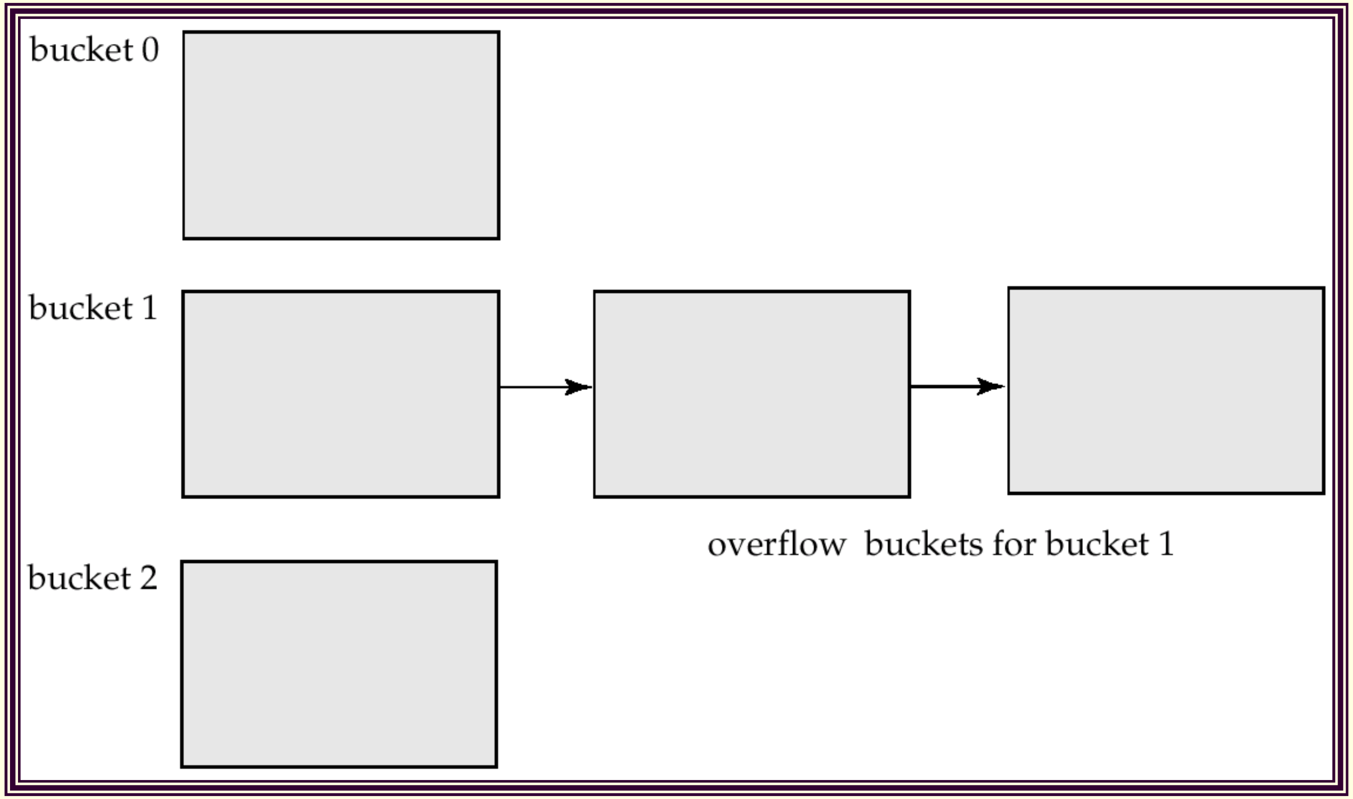
- Worst has function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- 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.

# Handling of Bucket Overflows

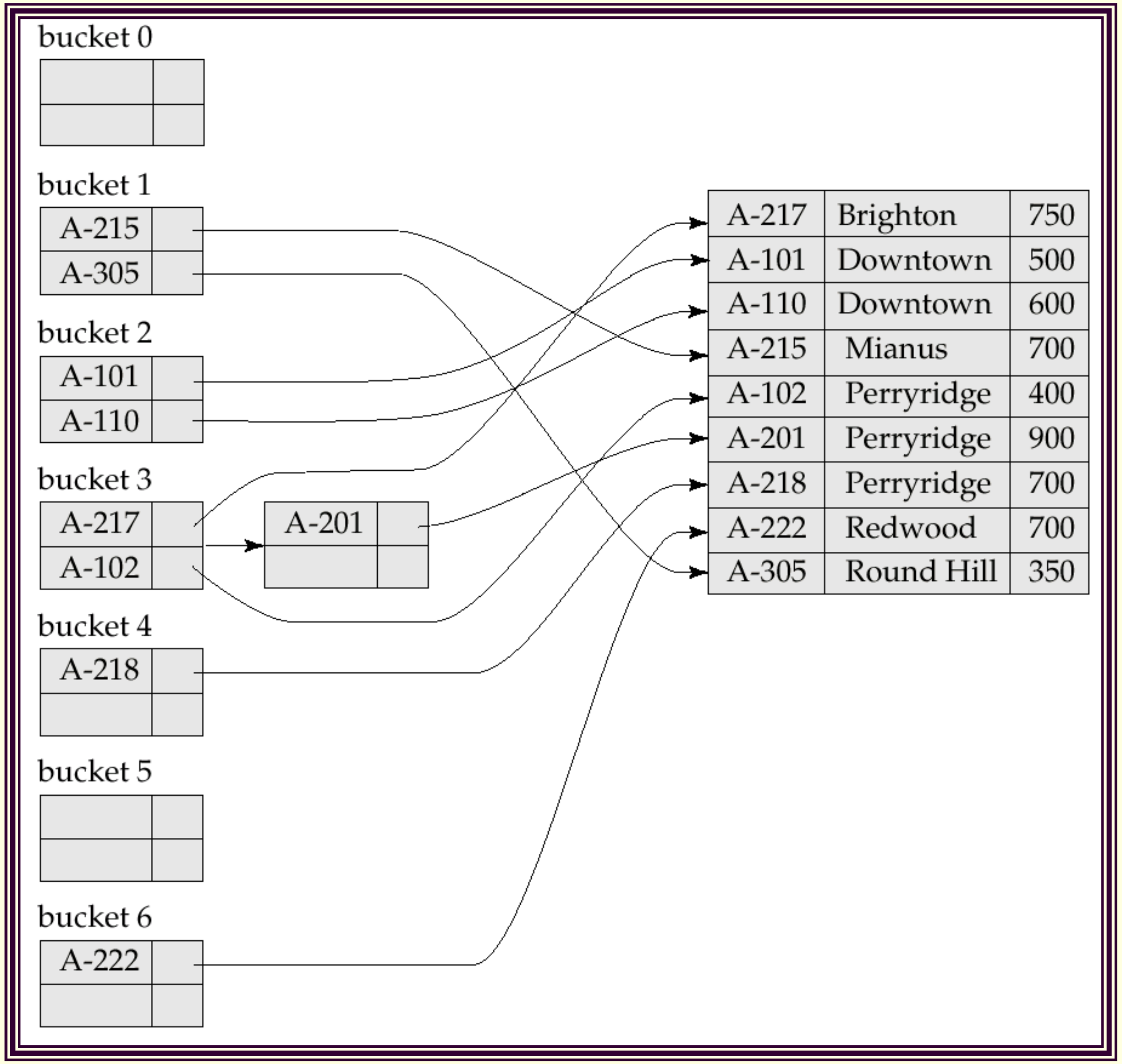
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow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

# Handling of Bucket Overflows (Cont.)

- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.



# E.g. of Secondary Hash Index



# Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *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.

# 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

# Secondary Index

- Provide access via non-key attributes
- Three data structures discussed here:
  - Linked lists: embedded pointers
  - Inverted lists: cross-index tables
  - B-Trees

# Index

- Primary mechanism to get improved performance on a database
- Persistent data structure, stored in database
- Many interesting implementation issues
  - But we are focusing on user/application perspective

# Index

| A | Pointer |
|---|---|
| cat | 1 |

| | |
|---|---|
| | 5 |
| | 2 |
| | 7 |
| | 6 |
| 9 | 4 |
| ... | |

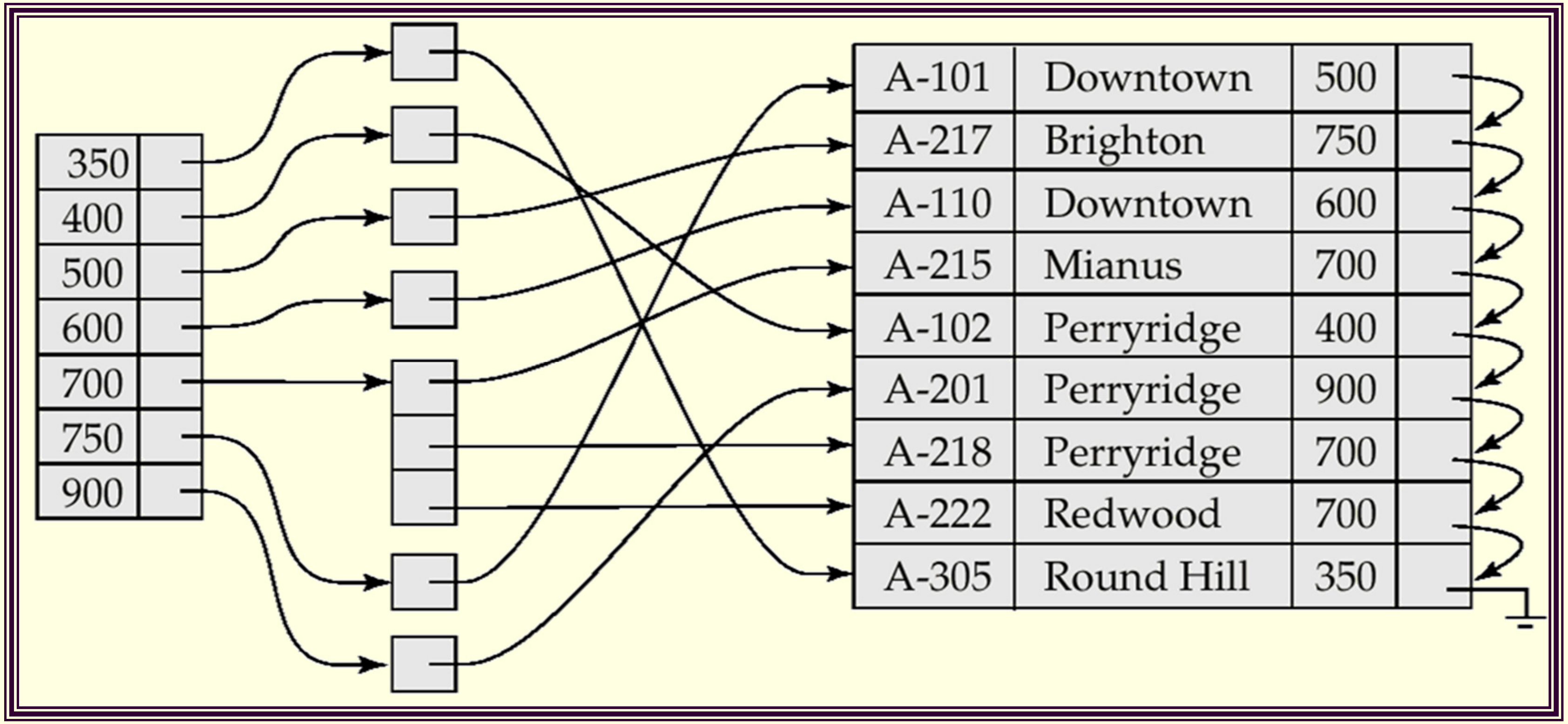
| A | B | Pointer |
|---|---|---|
| cat | 2 | 1 |
| cat | 2 | 5 |
| cat | 8 | 6 |
| cow | 1 | 3 |
| cow | 8 | 7 |
| dog | 5 | 2 |
| dog | 9 | 4 |
| ... | | |

Index on T.(A,B)

T

| | A | B | C |
|---|---|---|---|
| 1 | cat | 2 | ... |
| 2 | dog | 5 | ... |
| 3 | cow | 1 | ... |
| 4 | dog | 9 | ... |
| 5 | cat | 2 | ... |
| 6 | cat | 8 | ... |
| 7 | cow | 6 | ... |
| | ... | ... | ... |

# Secondary Index on *balance* field of *account*



# Primary and Secondary Indices

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- 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

# Index

- Difference between full table scans and immediate location of tuples
  - Orders of magnitude performance difference
- Underline Data Structures
  - Balance Trees (B Trees, B+ Trees)
  - Hash Tables

# Index – Key Field

```
Select sName
From Student
Where sID = 18942
```

- Many DBMS's build index automatically on `PRIMARY KEY` (and sometimes `UNIQUE`) attributes `sID`

# Index – Non-key Fields

```
Select sID
From Student
Where sName = 'Mary' And GPA > 3.9
```

- Index on **sName**
- Index on **GPA**
- Index on both **sName** and **GPA**

# Index – Join Tables

```
Select sName, cName
From Student, Apply
Where Student.sID = Apply.sID
```

Ex

# Question

- Consider the following query:

```
Select * from Apply, College
Where Apply.cName = College.cName and
    Apply.major = 'CS' and
    College.enrollment < 5000)
```

Which of the following indices could NOT be useful in speeding up query execution?

- Tree-based index on `Apply.cName`
- Hash-based index on `College.cName`
- Hash-based index on `Apply.major`
- Hash-based index on `College.enrollment`

Ex

# Question

- Consider the following query:

```
Select * from Student, Apply, College
Where Student.sID = Apply.sID and
      Apply.cName = College.cName and
      Student.GPA > 1.5 and
      College.cName < 'Cornell'
```

Suppose we are allowed to create two indexes, and assume all indexes are tree-based. Which two indexes would be most useful for speeding up query execution?

- `Apply.sID, Student.GPA`
- `Apply.cName, College.cName`
- `Student.sID, Student.GPA`
- `Student.sID, College.cName`

# Downsides of Index

- Extra Space
  - Marginal
- Index Creation
  - Medium
- Index Maintenance
  - Can often set off benefits

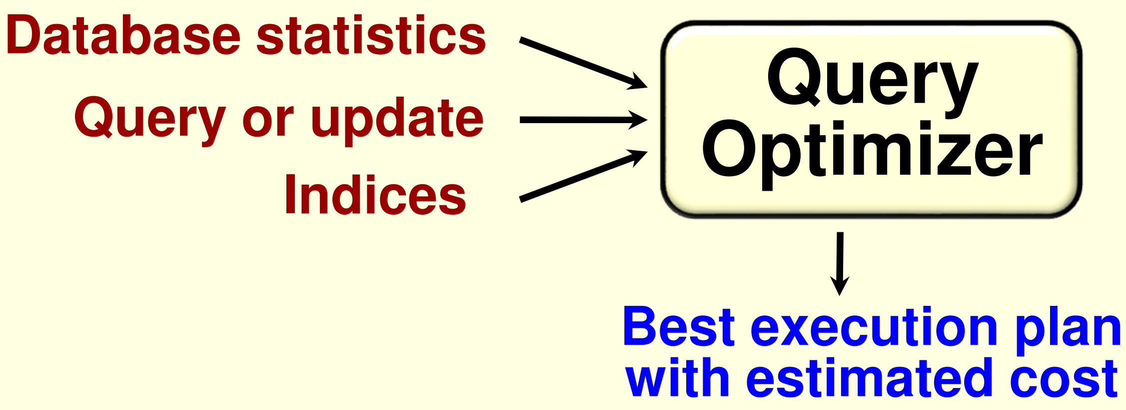
# When to Create an Index?

Benefit of an index depends on:

- Size of table (and possibly layout)
- Data distributions
- Query vs. update load

# Physical design advisors

- Input
  - Database (statistics) and workload
- Output
  - Recommended indices



# SQL - Create & Drop Index

- **Create Index IndexName on T(A)**
- **Create Index IndexName on T(A1,A2,…,An)**
- **Create Unique Index IndexName on T(A)**
- **Drop Index IndexName**

# Indices

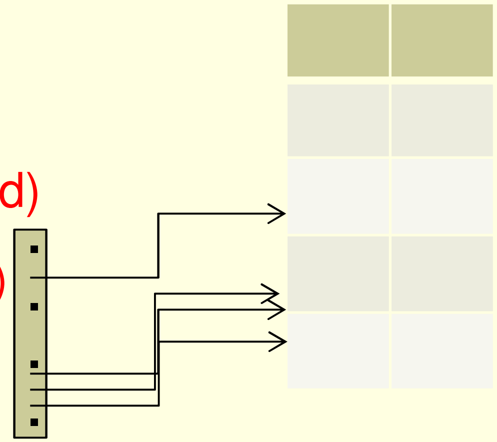
- SQL allows you to index database tables, making it possible to quickly seek to records without performing a full table scan first and thus significantly speeding up query execution.
- You can have many indexes (16 in MySQL) per table, and SQL also supports multi-column indices and full-text search indices.

# Selecting Records, e.g.

- Primary index (records ordered on a key field)

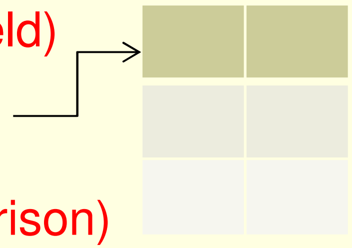
  T1 ← $\sigma_{empno='12345'}$(Employee) (single record)

  T1 ← $\sigma_{dno>='5'}$(Department) (multiple records)



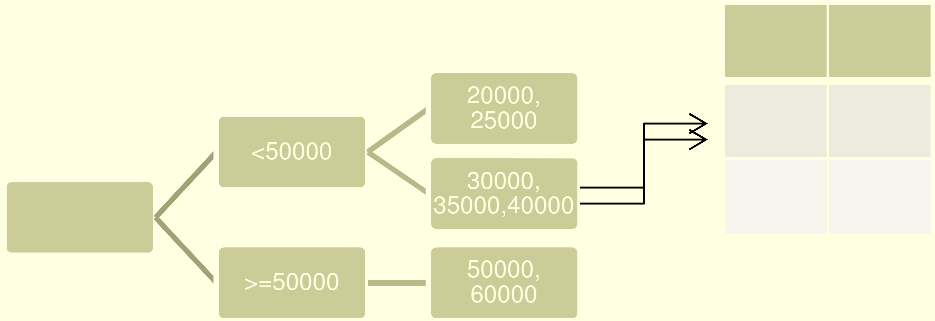
- Clustering Index (records ordered on a non key field)

  T1 ← $\sigma_{dno='5'}$(Employee) (multiple records)



- B⁺-Tree Index (secondary index on equality comparison)

  T1 ← $\sigma_{salary>=30000 \text{ and } salary<=35000}$(Employee) (multiple records)



# Cost of Operations

| | (a) Scan | (b) Equality | (c ) Range | (d) Insert | (e) Delete |
|---|---|---|---|---|---|
| (1) Heap | **BD** | **0.5BD** | **BD** | **2D** | **Search +D** |
| (2) Sorted | **BD** | **Dlog 2B** | **Dlog 2 B + # matches** | **Search + BD** | **Search +BD** |
| (3) Clustered | **1.5BD** | **Dlog F 1.5B** | **Dlog 2 1.5B + # matches** | **Search + D** | **Search +D** |
| (4) Unclustered Tree index | **BD(R+0.15)** | **D(1 +log F 0.15B)** | **Dlog F 0.15B + # matches** | **D(3 +log F 0.15B)** | **Search + 2D** |
| (5) Unclustered Hash index | **BD(R+0.125)** | **2D** | **BD** | **4D** | **Search + 2D** |

**B: The number of data pages**
**R: Number of records per page**
**D: (Average) time to read or write disk page**

*Several assumptions underlie these (rough) estimates!*