Indexing and Hashing

Basic Concepts

- Ordered Indices
- Hashing

- Index Definition in SQL
- Multiple-Key Access

aid	bname	balance
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- Example:

Account (aid, bname, balance) relation with unordered tuples

```
Select …
From Account
Where aid=…
```

```
Select …
From Account
Where bname=…
```

```
Select …
From Account
Where balance >= …
```

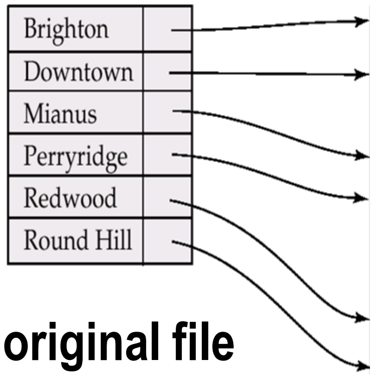
aid	bname	*balance*
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  Example: author catalog in library

- Search Key - attribute to 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:
  1. Ordered indices: search keys are stored in sorted order
  2. Hash indices: search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute (**equality**)
  - or records with an attribute value falling in a specified **range of values** (**between**).

- Access time
- Insertion time
- Deletion time

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.

Example of Relations

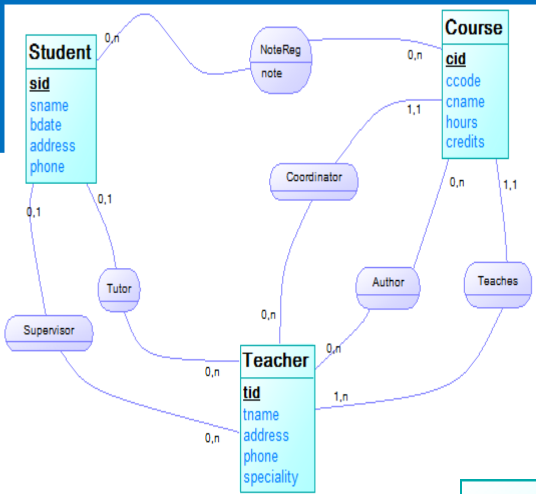
What are the relevant indices?

The *customer* Relation

customer-name	*customer-street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

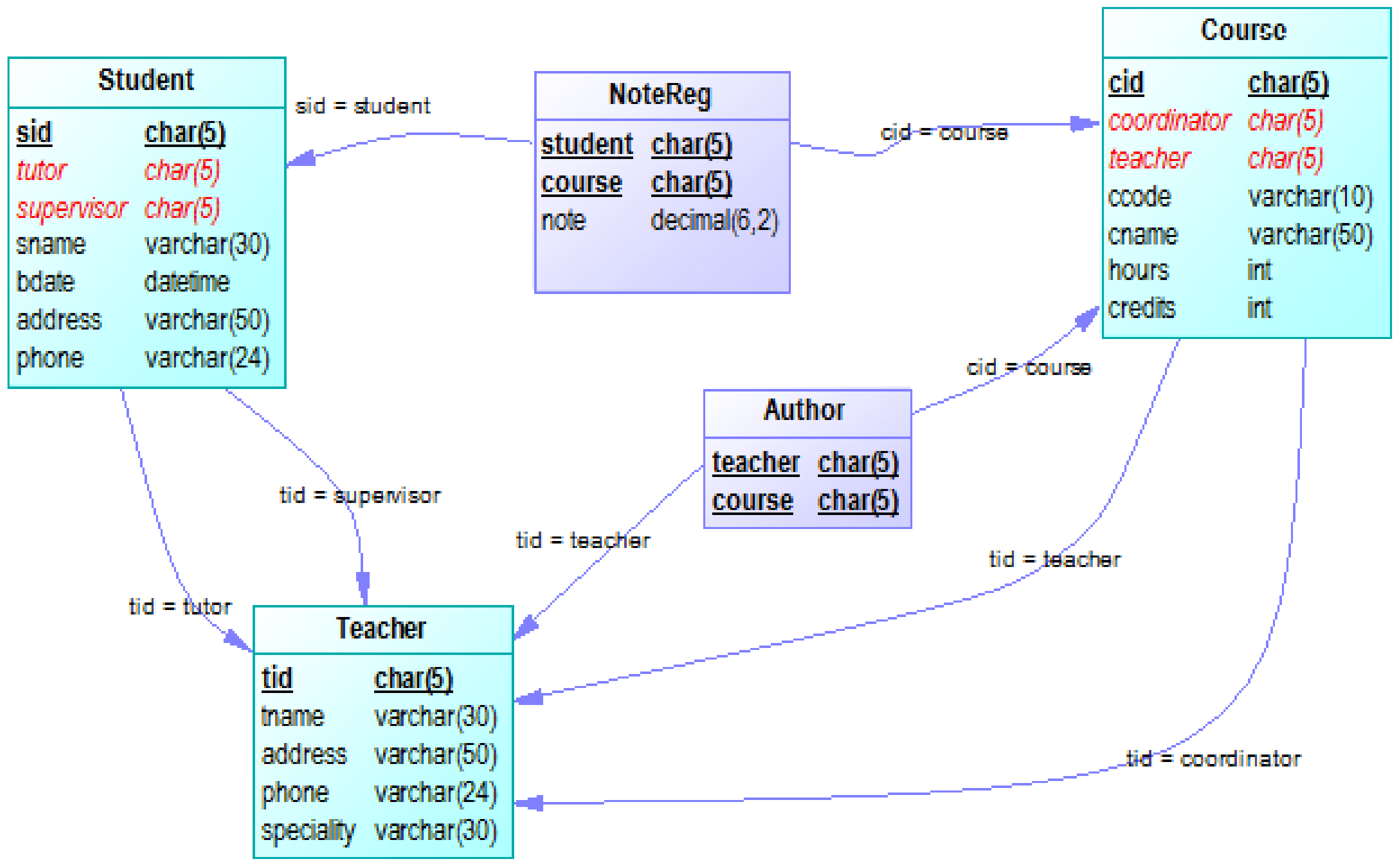
The *depositor* Relation

cname	aid
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305



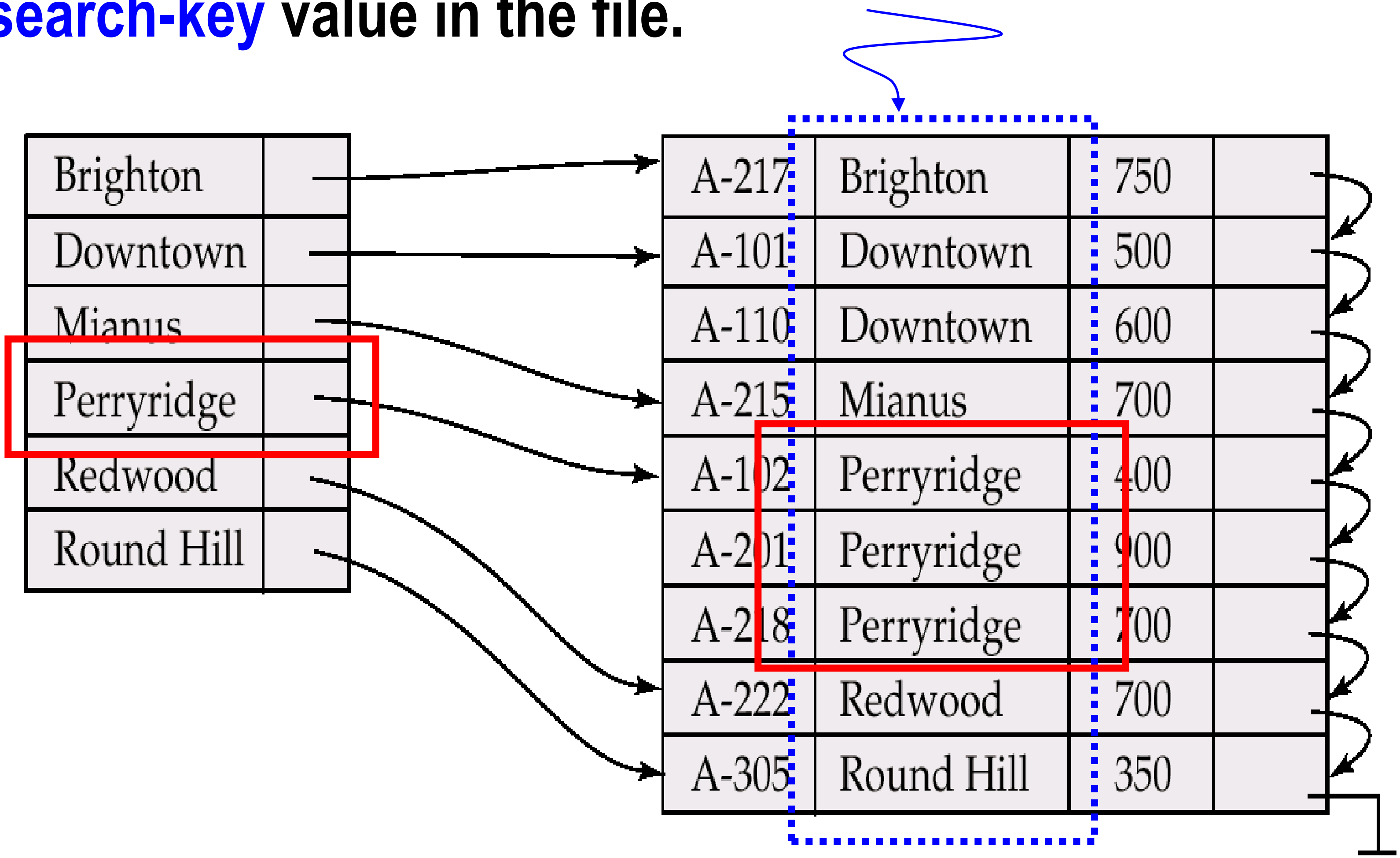
An example / Physical Data Model - Database Schema = Set of inter-related relations (tables)

What are the relevant indices?



Dense Index Files

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

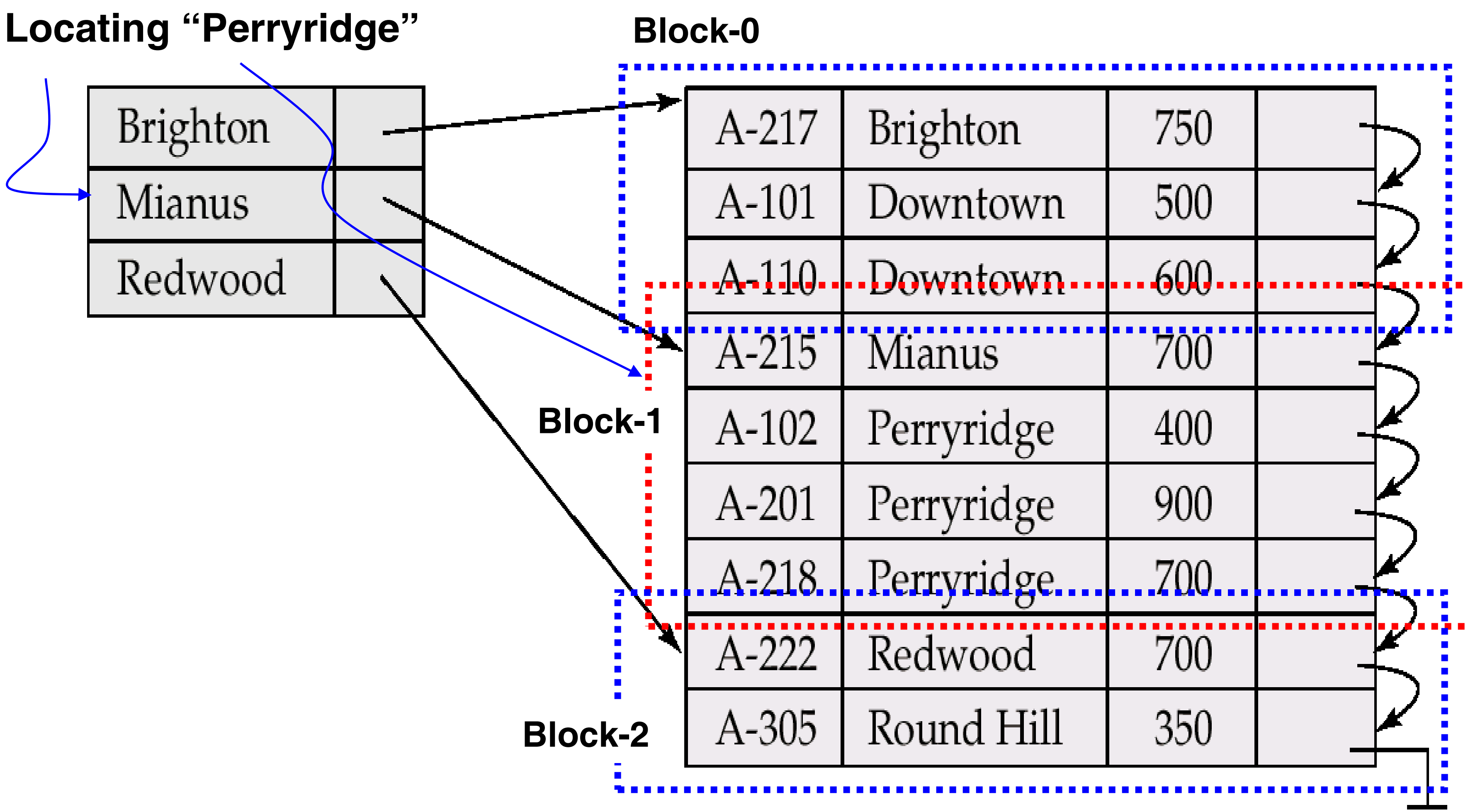


Sparse Index Files

- **Sparse Index:** contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- 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
- 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 least search-key value in the block.

Example of Sparse Index Files



Example: 1 block can contain 4 records!

Multilevel Index

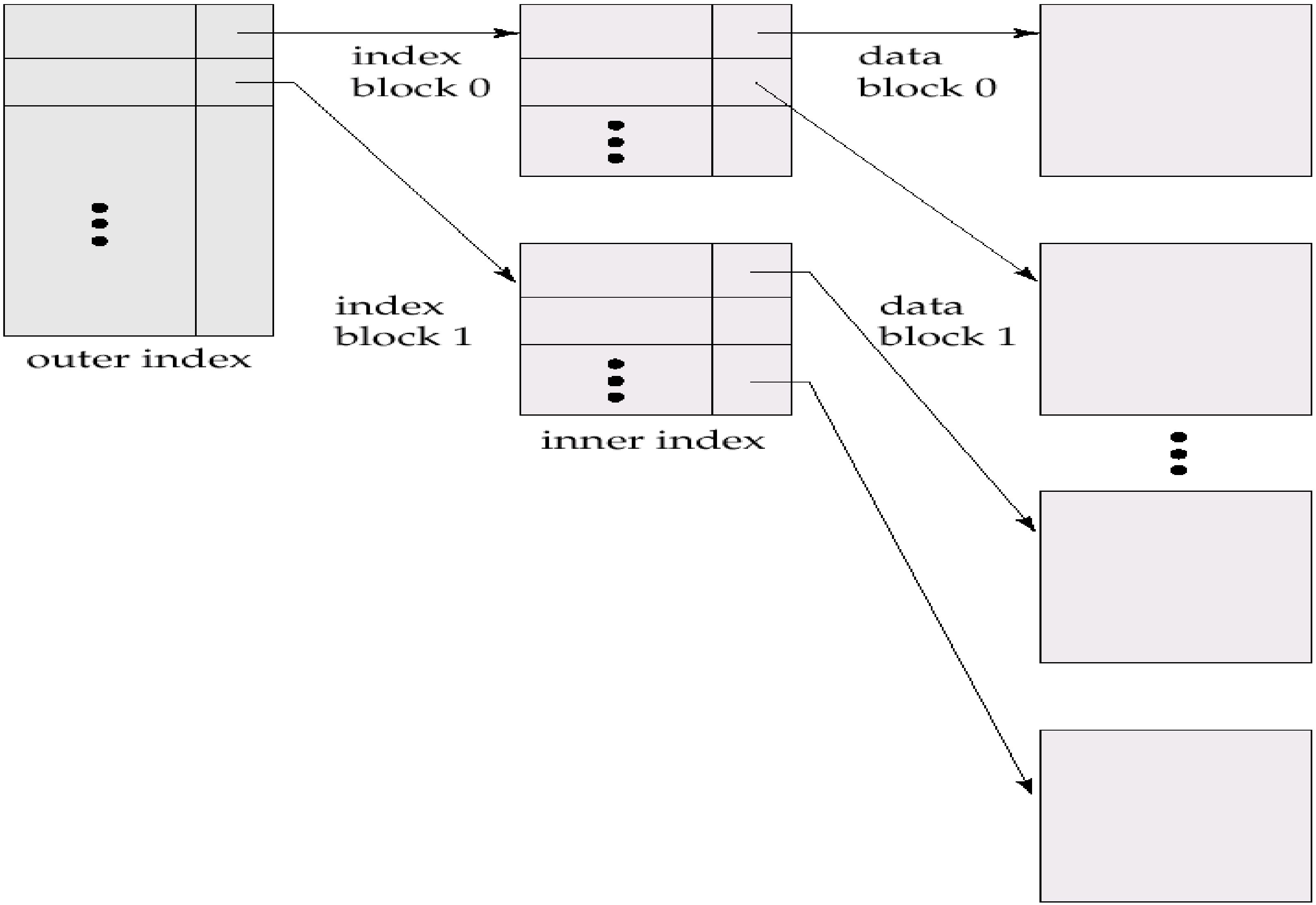
- If primary index **does not fit in memory**, access becomes expensive.

- 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.

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

Multilevel Index (Cont.)



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 deletion:

- Dense indices – deletion of search-key is similar to file record deletion.

Brighton	Block-0
Mianus	Block-1
Redwood	Block-2
…	

Block-0

…	Brighton	750
…	Downton	500
…	Downton	600

Block-1

…	Mianus	700
…	Perryridge	400
…	Perryridge	900
…	Perryridge	700

Block-2

…	Redwood	700
…	Round Hill	350

- 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).

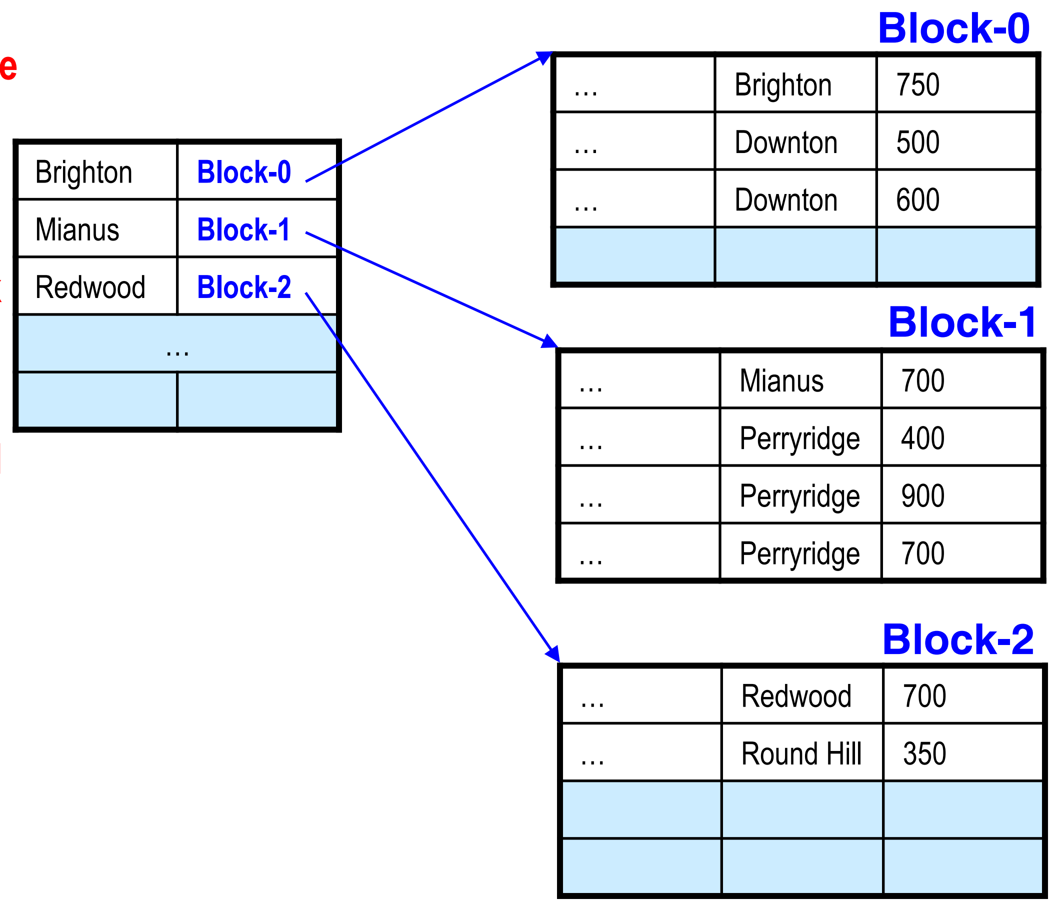
  If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

Single-level index insertion:

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

  In this case, the first search-key value appearing in the new block is inserted into the index.



Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

Secondary Indices

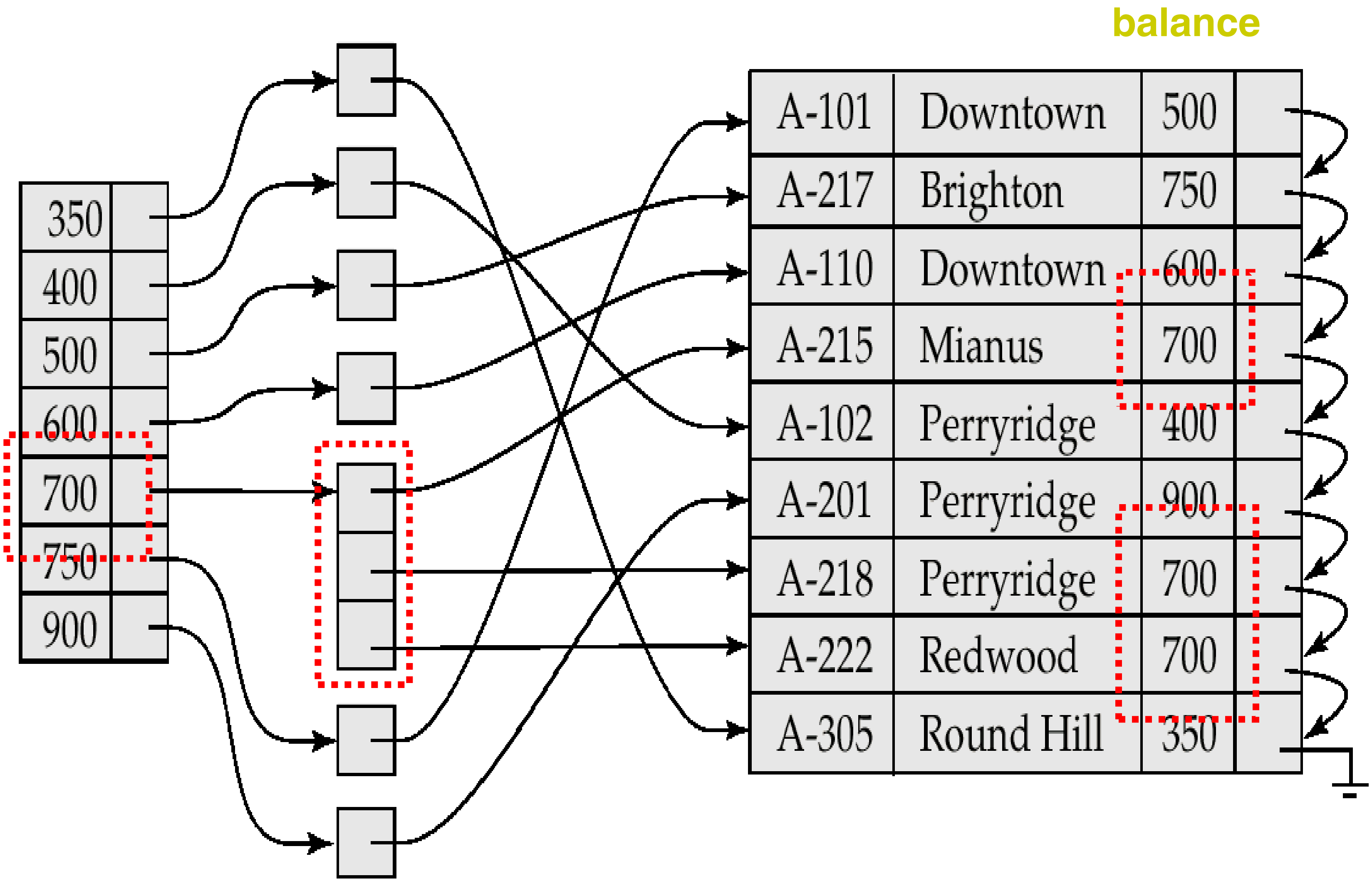
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.

  Example 1: In the account database stored sequentially by account number, we may want to find all accounts in a particular branch

  Example 2: as above, but where we want to find all accounts with a specified balance or range of balances

- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

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$_{\text{frais}}$ 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

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.

$$\{K\} \xrightarrow{h} \{B\}$$

- 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.

Example of Hash File Organization (Cont.)

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

- **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) = 5   h(Round Hill) = 3   h(Brighton) = 3**

Example of Hash File Organization

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

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.

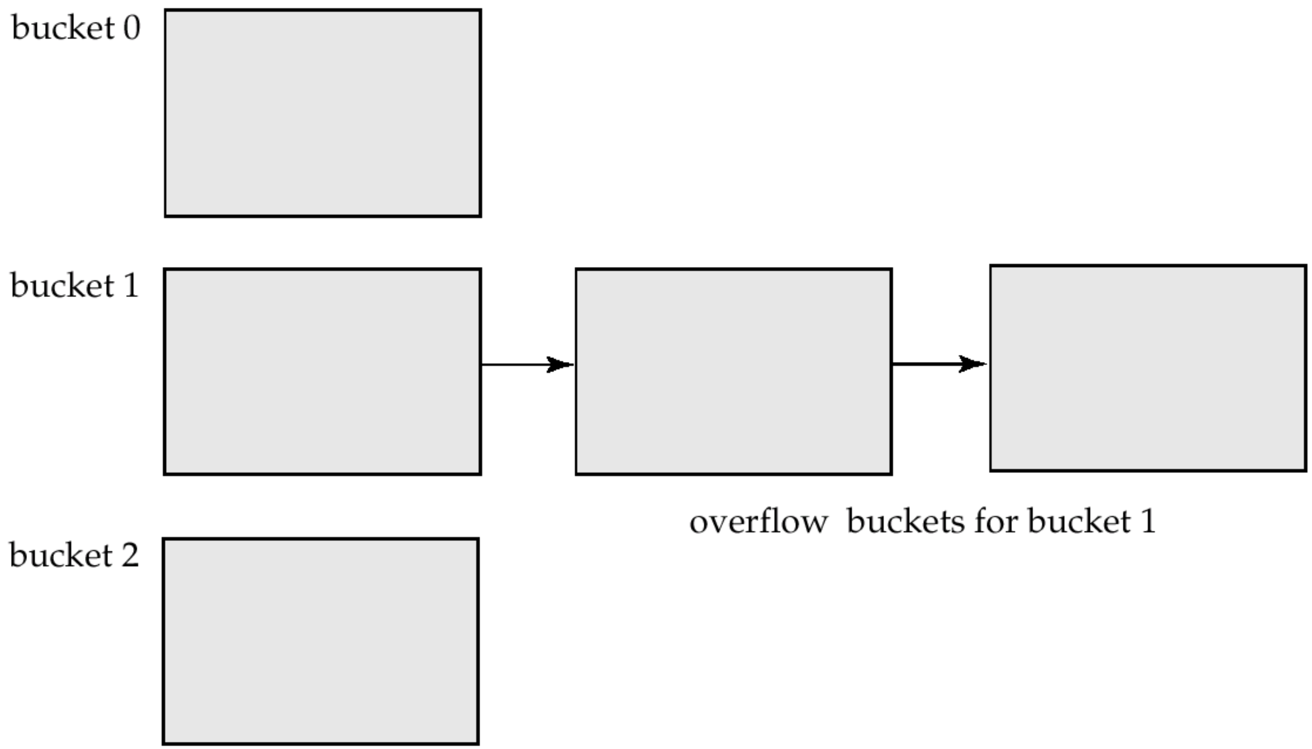
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

Handling of Bucket Overflows

- Bucket 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.
- Above scheme is called closed hashing.
  - An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.



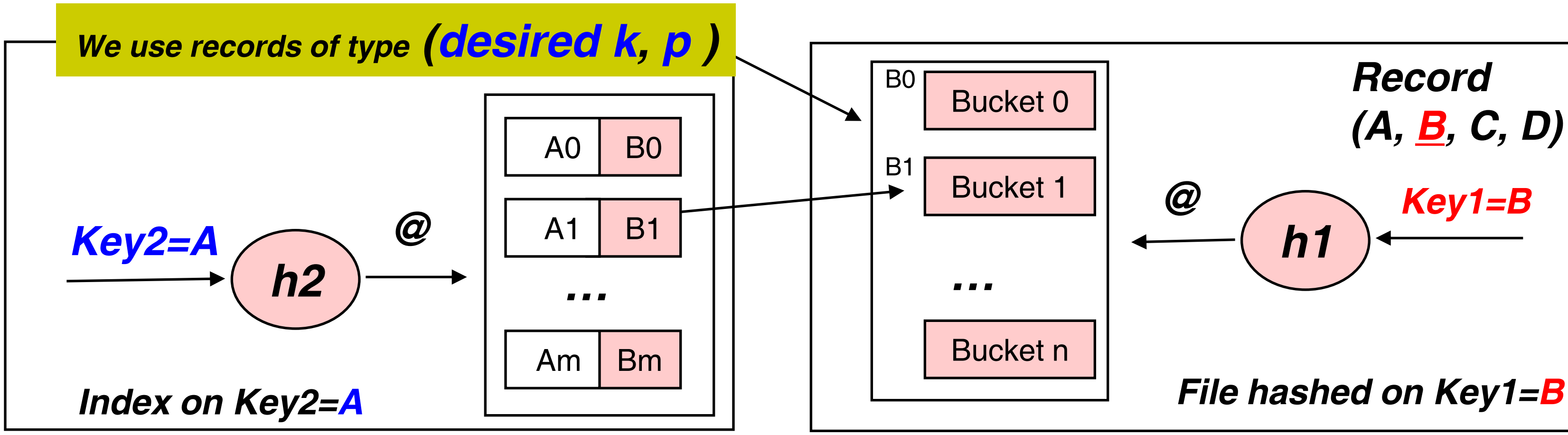
Hash Indices

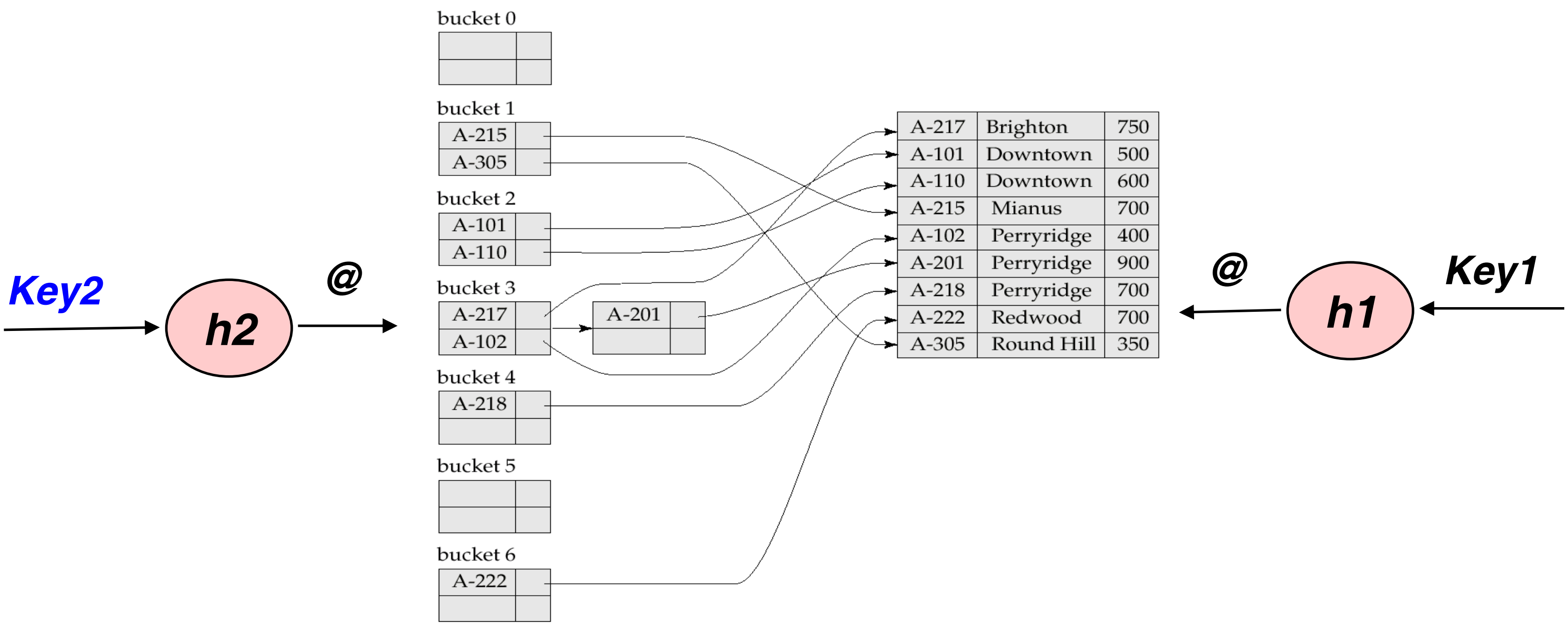
- Hashing can be used not only for file organization, but also for index-structure creation.

- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.

- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index





Index Definition in SQL

- Create index …
- Drop index ….

Index Definition in SQL

- **Create an index:**

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

  E.g.: create index ***b-index*** on ***branch(branch-name)***

- **Use** create unique index **to indirectly specify and enforce the condition that the search key is a candidate key.**
  - **Not really required if SQL** unique **integrity constraint is supported**

- **To drop an index:**

  drop index **<index-name>**

Multiple-Key Access

- Use multiple indices for certain types of queries.

Example:

```
select    account-number
from      account
where     branch-name = "Perryridge"
      and     balance = 1000
```

- Possible strategies for processing query using indices on single attributes:

1. Use index on branch-name to find accounts with balances of $1000; test branch-name = "Perryridge".
2. Use index on balance to find accounts with balances of $1000; test branch-name = "Perryridge".
3. Use branch-name index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on balance. Take intersection of both sets of pointers obtained.

Indices on Multiple Attributes

Suppose we have an index on combined search-key

(branch-name, balance).

- With the where clause
  where branch-name = "Perryridge" and balance = 1000
  the index on the combined search-key will fetch only records that satisfy both conditions.
  Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  where branch-name = "Perryridge" and balance < 1000
- But cannot efficiently handle
  where branch-name < "Perryridge" and balance = 1000
  May fetch many records that satisfy the first but not the second condition.

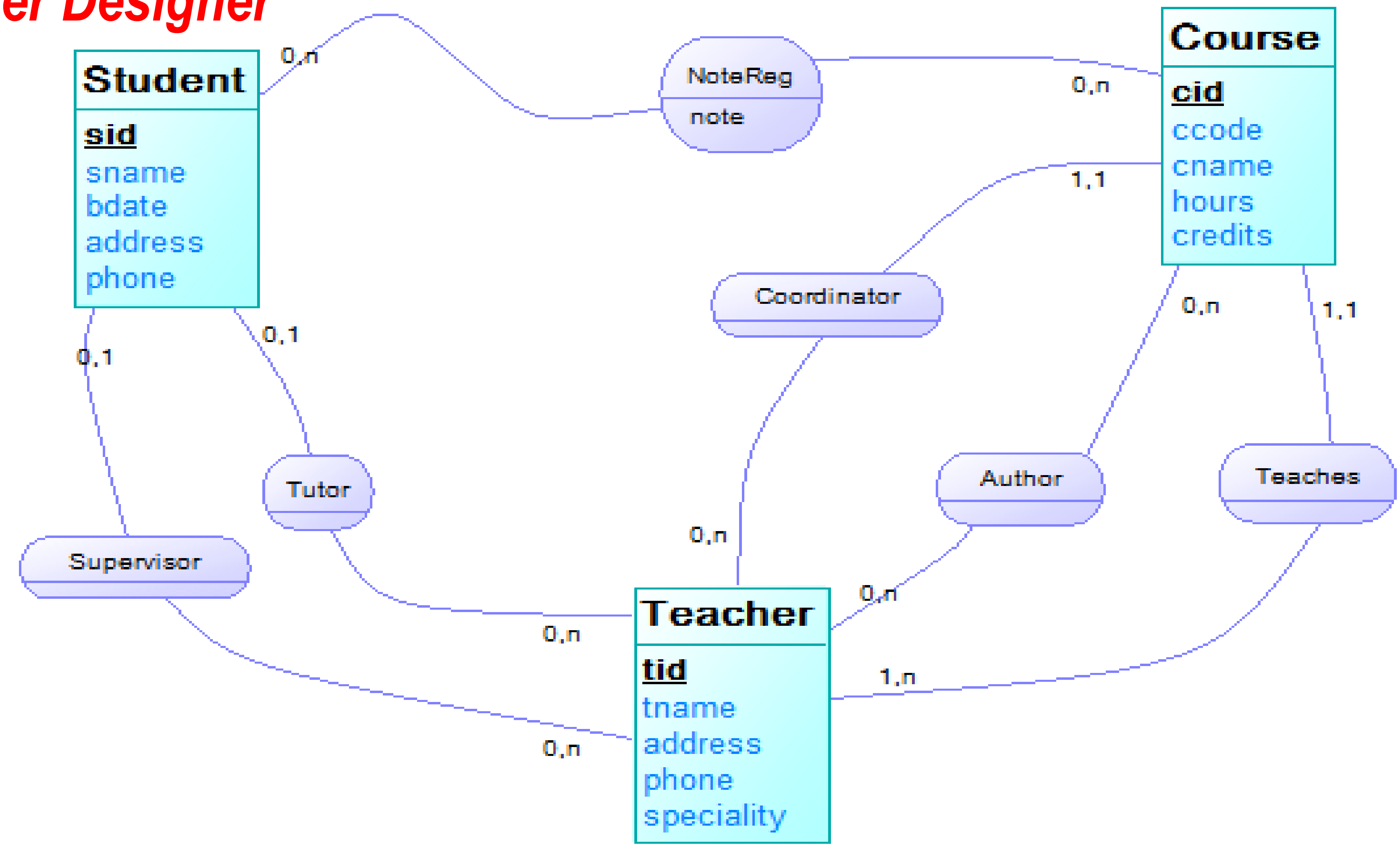
Use case: Database about students

- Considering the database schema
- Indicate relevant indices

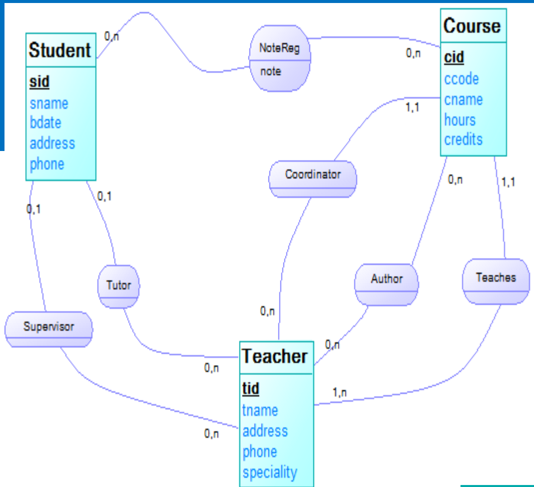
- In practice; MS-SQLServer
  - SQL Create tables
  - Create the considered indices
  - Add/delete indices

An example / Student management data

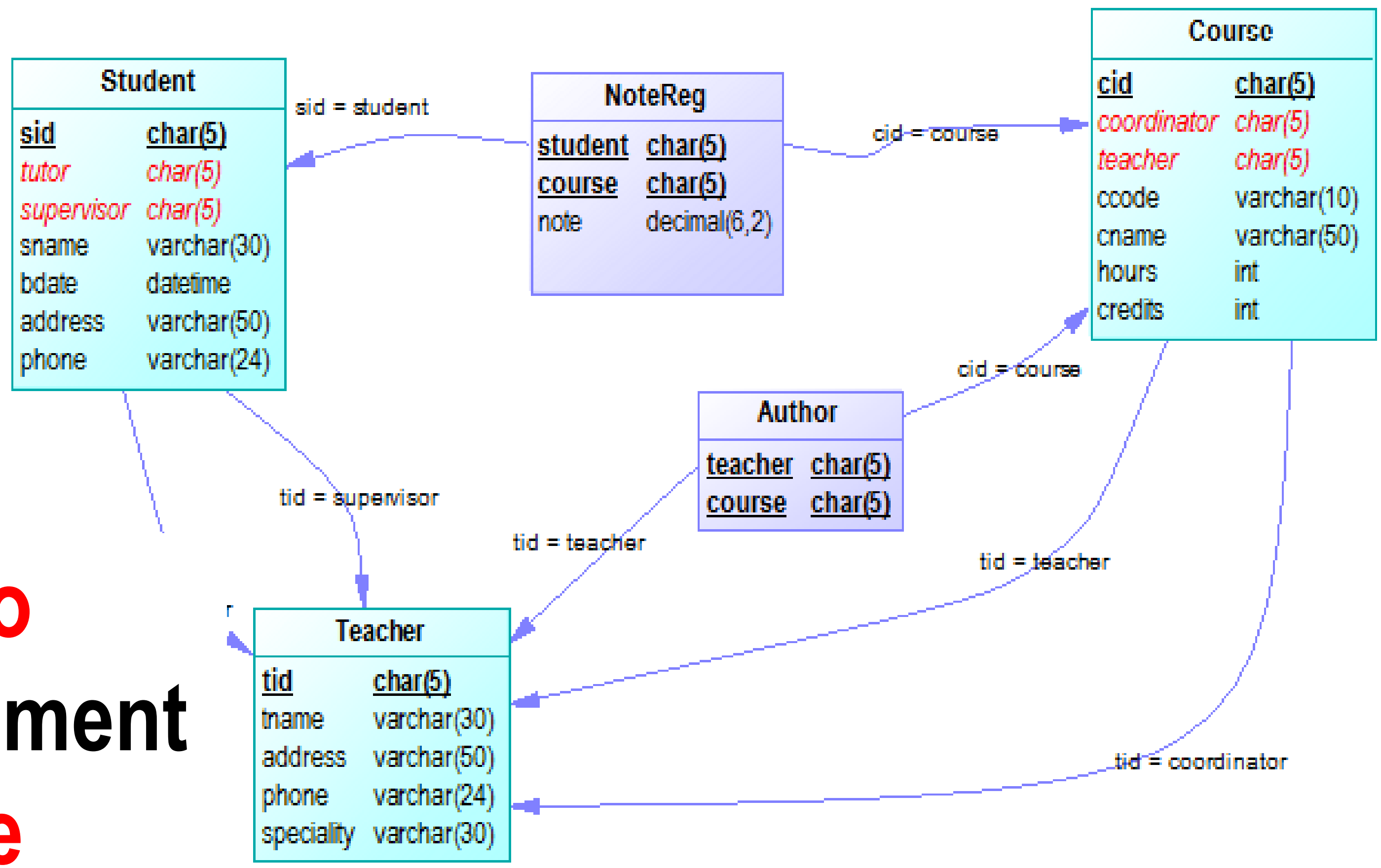
Power Designer



An example / Physical Data Model -
Database Schema = Set of inter-related relations (tables)



Teacher (tid, tname, address, phone, speciality)
Course (cid, cname, hours, credits, *#coordinator, #teacher*)
Author (#teacher, #course)
Student (sid, sname, bdate, address, phone, *#tutor* , *#supervisor*)
NoteReg (#student, #course, note)



What want to create/implement the database

Create tables….

Teacher (tid, tname, address, phone, speciality)
Course (cid, cname, hours, credits, #coordinator, #teacher)
Author (#teacher, #course)
Student (sid, sname, bdate, address, phone, #tutor , #supervisor)
NoteReg (#student, #course, note)

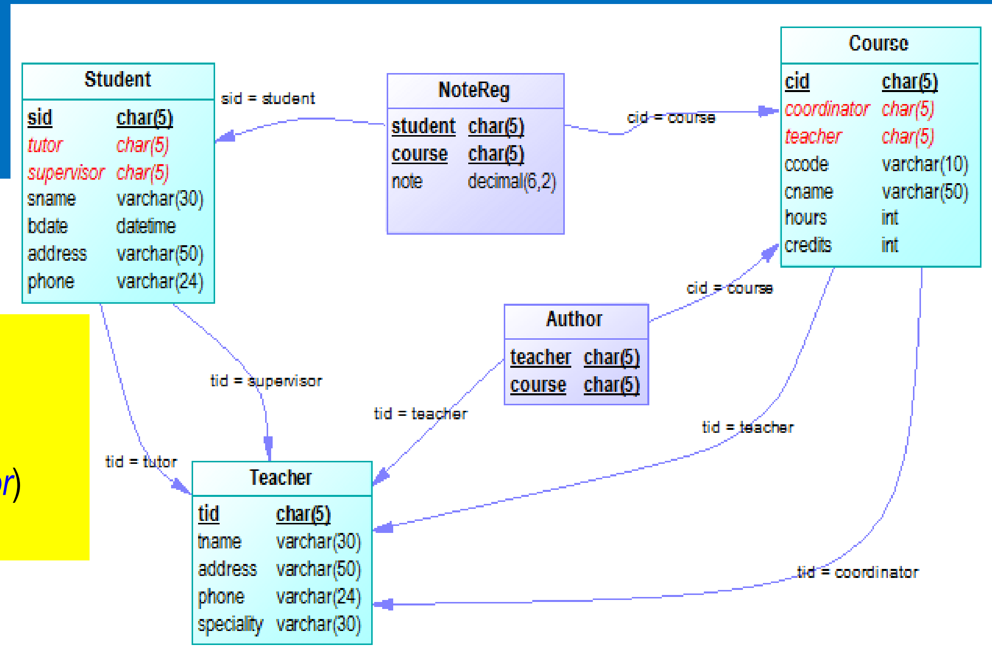
```
create table Teacher (
  tid          char(10)       not null,
  tname        varchar(30)    null,
  address      varchar(50)    null,
  phone        varchar(24)    null,
  speciality   varchar(30)    null,
  primary key (tid)
)
create table Course (
  cid          char(10)       not null,
  coordinator  char(10)       not null,
  teacher      char(10)       not null,
  cname        varchar(50)    null,
  hours        int            null,
  credits      int            null,
  primary key (cid),
  foreign key (teacher)  references Teacher (tid),
  foreign key (coordinator) references Teacher (tid)
)
create table Author (
  teacher      char(10)       not null,
  course       char(10)       not null,
  primary key (teacher, course),
  foreign key (course)  references Course (cid),
  foreign key (teacher)  references Teacher (tid)
)
```

The primary key constraint implemented via a cluster index

```
create table Student (
  sid          char(10)       not null,
  tutor        char(10)       not null,
  supervisor   char(10)       not null,
  sname        varchar(30)    null,
  bdate        datetime       null,
  address      varchar(50)    null,
  phone        varchar(24)    null,
  primary key (sid),
  foreign key (supervisor)  references Teacher (tid),
  foreign key (tutor) references Teacher (tid)
)
create table NoteReg (
  student      char(10)       not null,
  course       char(10)       not null,
  note         numeric(6,2)   null,
  primary key (student, course),
  foreign key (course) references Course (cid),
  foreign key (student) references Student (sid)
)
```

What are the relevant indices

Teacher (tid, tname, address, phone, speciality)
Course (cid, cname, hours, credits, #coordinator, #teacher)
Author (#teacher, #course)
Student (sid, sname, bdate, address, phone, #tutor , #supervisor)
NoteReg (#student, #course, note)



PRIMARY KEY and UNIQUE constraints:

When you create a PRIMARY KEY constraint, a unique clustered index on the column or columns is automatically created if a clustered index on the table does not already exist and you do not specify a unique nonclustered index

Create the relevant indices?

create index course_FK on **Author** (***course*** ASC)
~~create index teacher_FK on **Author** (***teacher*** ASC)~~ ✗
create index Teaches_FK on **Course** (***teacher*** ASC)
create index Coordinator_FK on **Course** (***coordinator*** ASC)
create index course_FK on **NoteReg** (***course*** ASC)
~~create index student_FK on **NoteReg** (***student*** ASC)~~ ✗
create index Tutor_FK on **Student** (***supervisor*** ASC)
create index Sup_FK on **Student** (***tutor*** ASC)