PURDUE UNIVERSITY

CS34800
Information Systems

Indexing & Hashing

Prof. Chris Clifton

31 October 2016

First: Triggers - Limitations

- Many database functions do not quite work as expected
  - One example: Trigger on a table that modifies that table
- Often published "work-arounds"
- You may run into one of these on Project 3 Question 5b
  - Will send out a "hint" this afternoon
  - This will only be 5 points (i.e., we only expect A-level students to get this part.)

Indexing and Hashing

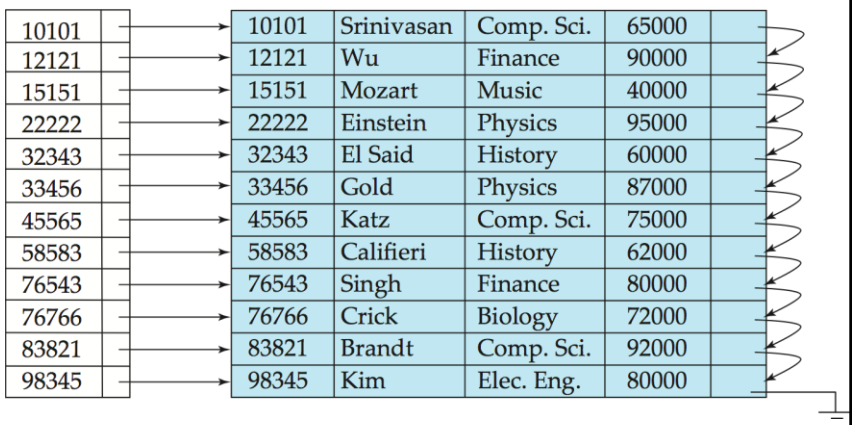
- Goal: Faster access to data
  - Faster than scanning the whole table
- Search Key: attribute/column for which faster search enabled
  - Not the same as keys for database design
- Index: Tree structure allowing faster search
  - Logarithmic time
- Hashing: Group data into "buckets" based on value of search key
  - If all goes well, constant time access

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.

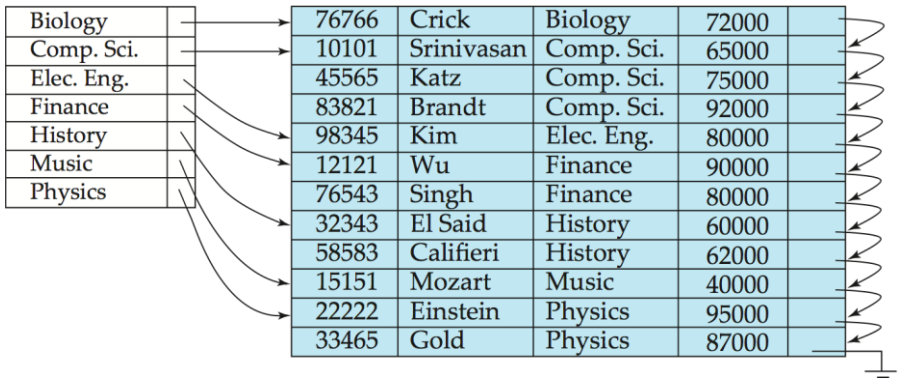
Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



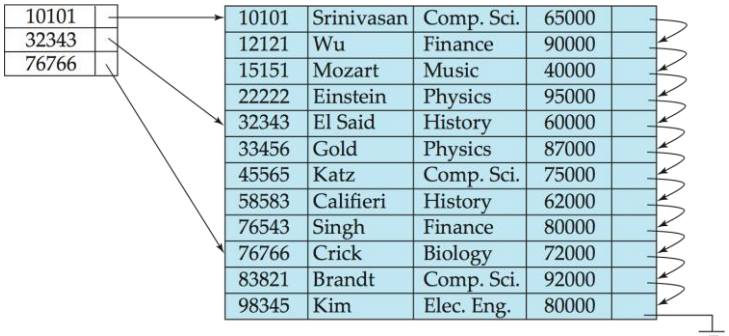
Dense Index Files (Cont.)

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



Sparse Index Files

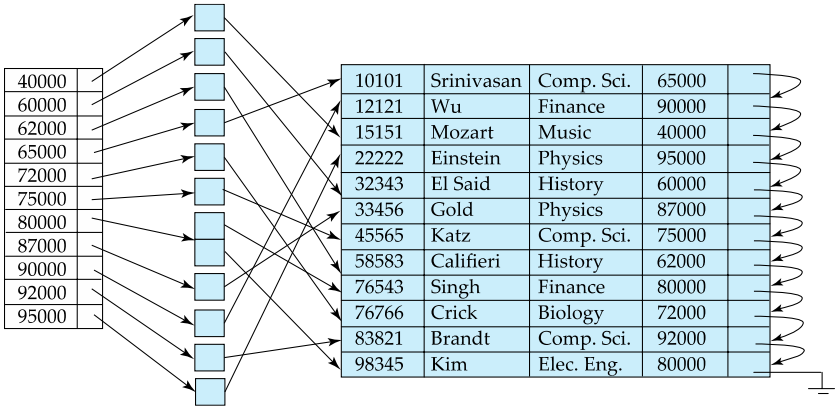
- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points



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: as above, but where 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 Example



Secondary index on *salary* field of *instructor*

- 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.
- 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

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.

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_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(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3

Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key (see previous slide for details).

Hash Functions

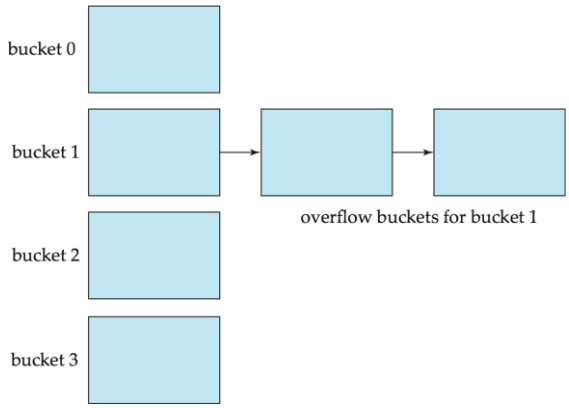
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
-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



Creating Indexes

- create index <name> on <relation> (<attribute_list>)
  - create index students_i_name on students (lastname);
- Can specify type of index
  - create bitmap index students_i_id on students(StudentID);
- Also delete: drop index student_i_name;
- Oracle: Index automatically created for primary key or unique constraint

Bitmap Index

- Similar in concept to hashing
  - Key values represented as bits in a (long) vector
  - Particularly good when few possible key values
- Supports easy and/or operations in queries
  - lastname = 'Clifton' AND salary > $100k
- More expensive to update