CMSC 424 – Database design
Lecture 15
Hashing
Query processing

Mihai Pop

Admin issues

- Sample midterm on website later today
- No office hours on Wednesday – email me if you need help preparing for midterm.
- Project questions?

Add'l. notes

- B+-trees can be used as an index structure (as described before)
- B+-trees can also be used to organize the file
  - leaves and internal nodes contain entire records
  - fan-out is limited (due to large size of records)
- Even B+-tree structures may need to be rebuilt to enhance performance (e.g. due to insertions/deletions order of blocks on disk becomes inefficient)
- Secondary indices may need to be updated every time we modify a B+-tree file (order of records may change even though records did not change)
  - solution: secondary indices point to the primary search key instead of the record – slower access but no need for frequent updates

More…

- Hash-based Indexes
  - Static Hashing
  - Dynamic Hashing
    - Read on your own.
  - Linear Hashing
- Grid-files
- R-Trees
- etc…

Unordered Indexes: Static Hashing

- A bucket is a unit of storage containing one or more records (a bucket is typically a disk block or several disk blocks)
- 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.

Hashed File

- divide the set of blocks into *buckets*
- devise a hashing function that maps each key value into a bucket

  V: set of key values

  B: number of buckets

  H: hashing function H: V-> {0,1,2,…,B-1}

Example: V: 9 digit SS#, B: 1000, H: key MOD 1000

- search for, insert, delete, modify a key k do
  - H(k) to get the bucket number
  - search sequentially in the bucket (heap organization within each bucket)
- selection of H: almost any function that generates "random" numbers in [0,B-1]
  - try to distribute evenly the keys into the B buckets
  - H(key)=3 (constant function) is bad- all records go to the 3rd bucket
  - rule of thumb for MOD: prime number
- collisions: two or more key values go to the same bucket
  - too many collisions increases the search time degrades performance

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. .
  - From book: hash function for strings:

$$s[0]\cdot 31^{n-1}+s[1]\cdot 31^{n-2}+\ldots+s[n-1]$$

Hashed File

- hash table: holds the physical address of the buckets

<u>Example:</u>

EMP(ename,sal)   H(sal): sal MOD 3

- overflow may occur for the following reasons:
  - too many records
  - poor hashing function
  - skewed data (too many values hash to the same bucket)

- overflow is handled by one of the two methods:
  - <u>*chaining*</u> of multiple blocks in a bucket
  - <u>*open addressing*</u>: if the hashed bucket H(k) is full, put it in H(k)+1. If also full, in H(k)+2, etc. NOT USEFUL FOR DATABASES
  - <u>*double hashing*</u>: hash once to H(k). If full, hash again with another H'(k). If still full, then apply any of the above methods

- performance depends on the <u>*loading factor*</u>=# of records/(B*f) where f is the number of keys in a block
  - rule of thumb: when loading factor too high, double B and rehash

Search Cost of a Hashed File

- assume the hash table is in main memory
  - successful search: best case 1 block, worst all chained bucket blocks, average half of worst
  - unsuccessful search: best, worst, average all chained bucket blocks
  - for loading factor of about 90% and a good hashing function average is 1.2 blocks

- Advantage of hashing: very fast for *exact* queries

- Disadvantage of hashing : since the records are not sorted in any order, it cannot do *range* queries

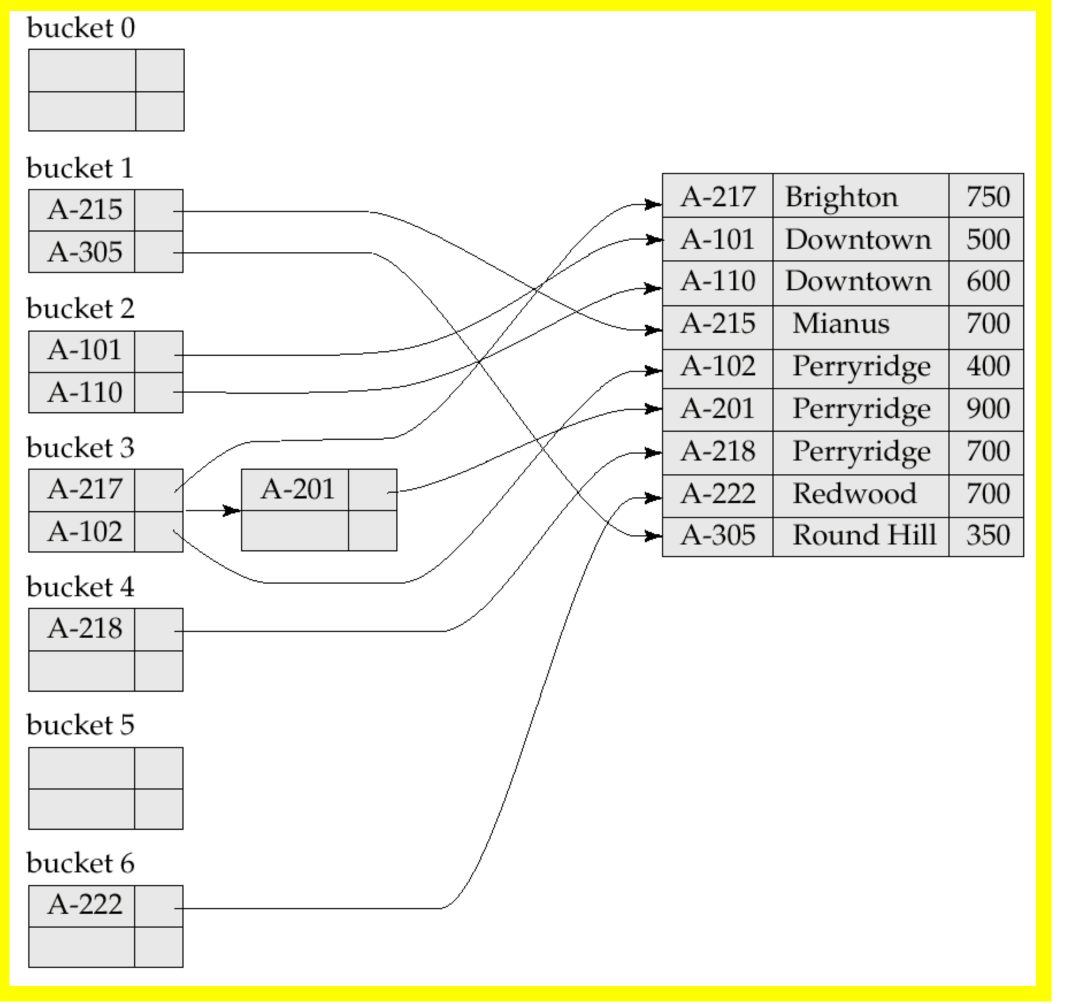
Hash Indices

- 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



Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses
- Databases grow over 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 distance 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 and
  - impossible in 7-24 databases

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)*

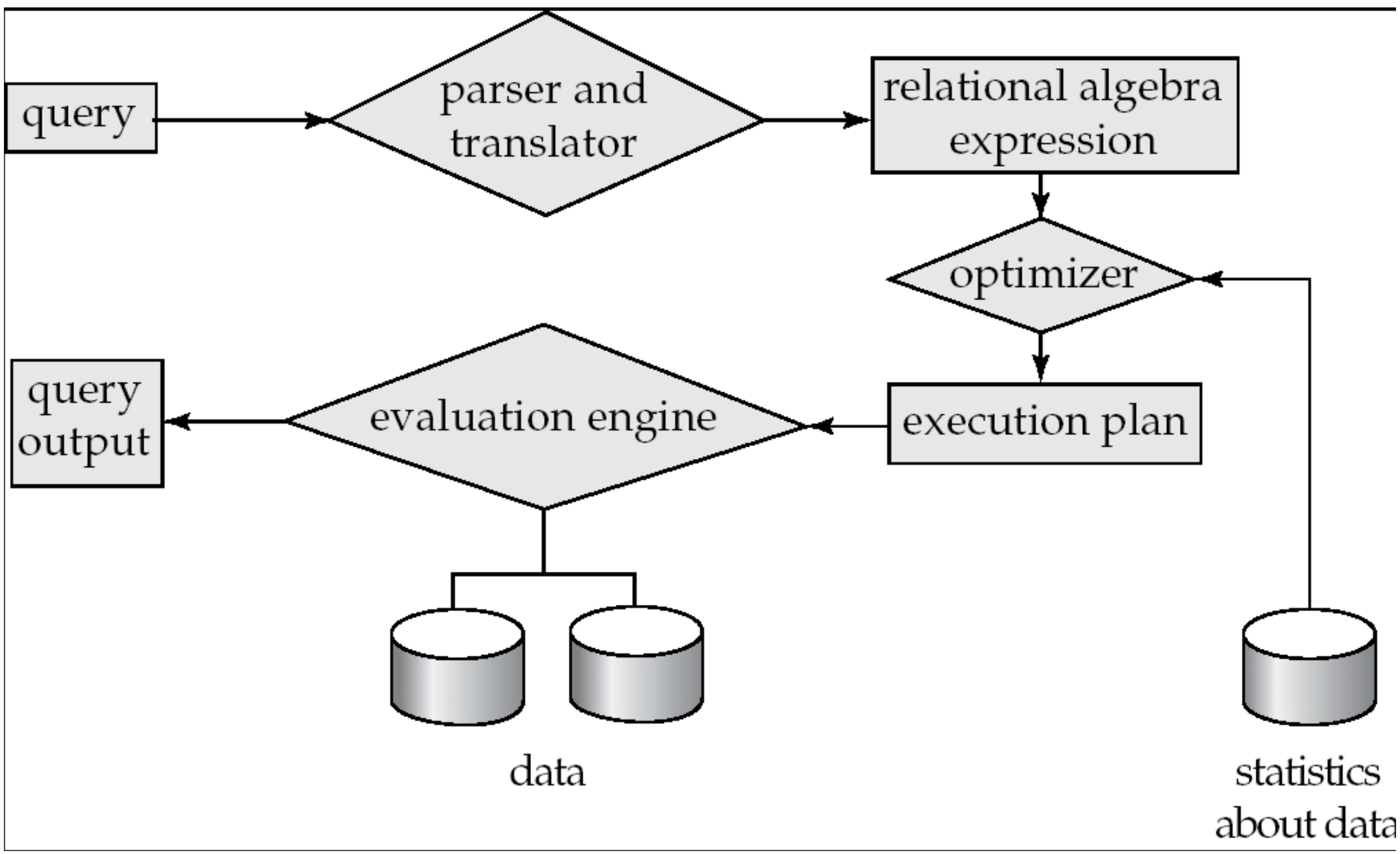
- Use create unique index to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL unique integrity constraint is supported

- To drop an index

  drop index <index-name>

13. Query Processing

- Steps
  1. parsing & translation: SQL ==> Internal relational form
  2. optimization: pick amongst several the best plan
  3. evaluation of the selected plan



SQL is not "query friendly"

- SQL command tells you WHAT to do not HOW to do it
- select balance
  from account
  where balance < 2500
- can be written as either:
  - $\sigma_{\text{balance}<2500}(\pi_{\text{balance}}(\text{account}))$
  - $\pi_{\text{balance}}(\sigma_{\text{balance}<2500}(\text{account}))$
- Picking among these depends on many factors (e.g. is there an index on balance?, what type of index is used?...)
- A query-evaluation plan also records additional information:

$$\pi_{\text{balance}}(\sigma_{\text{balance}<2500\,;\,\text{using index 1}}(\text{account}))$$

Query Processing

- <u>Cost parameters (*some are easy to maintain some are very hard*)</u>

– statistical info maintained in the system's catalog

n(r) =   number of tuples in the relation r

b(r) =   number of blocks containing tuples of relation r

s(r) =   average size of a tuple of relation r

f(r) =   blocking factor of r, I.e. the number of r tuples that fit in a block

V(A,r) = number of distinct values of attribute A in r
=   n(r) if A is a key

SC(A,r) = average selectivity of attribute A in r (# of tuples selected per value of A)
= 1   if A is a key
= n(r) / V(A,r)   otherwise

Query processing

min(A,r) = minimum value of attribute A in r

max(A,r) = maximum value of attribute A in r

- Two important computations
  - I/O cost of each operation
    - Number of blocks accessed
    - Number of seeks
  - the size of the result

Selection / Projection File Scan

- A1: search for equality: R.A=c cost (seq. search rel. sorted)

  $= b(r)/2 + \lceil SC(A,r)/f(r) \rceil - 1$ average successful
  $= b(r)/2$ average unsuccessful
- A2: (binary search)

  $= \lceil \log b(r) \rceil + \lceil SC(A,r)/f(r) \rceil - 1$ average successful
- Size of the result: $n(\sigma(R.A=c)) = SC(A,r) = n(r) / V(A,r)$
- search for inequality: R.A>c
  - cost (file unsorted) = b(r)
    (sorted on A) = b(r)/2+ b(r)/2 (if we assume that half of the tuples qualify)

  - size of the result: $n(\sigma(R.A>c)) = [max(A,r)-c] * n(r) / [max(A,r) - min(A,r)]$
- projection on A
  - cost as above
  - size of the result: $n(\pi(R,A)) = V(A,r)$