Today's Lecture

• Linear Hashing. Section 11.3
• Overview of query evaluation
• Evaluating Relational Operators
  – Select
  – General Select Conditions
  – Project Operation

Linear Hashing

• This is another dynamic hashing scheme, an alternative to Extendible Hashing.
• LH handles the problem of long overflow chains without using a directory, and handles duplicates.
• Idea: Use a family of hash functions $h_0, h_1, h_2, ...$
  – Each function's range is twice the previous
  – $h_i(key) = h(key) \mod(2^N)$; $N = \text{initial # buckets}$
  – $h$ is some hash function (range is not 0 to N-1)
  – Let $d_0$ be the number of bits to represent N
  – $N = 2^{d_0}$, $h_i$ consists of applying $h$ and looking at the last $d_i$ bits, where $d_i = d_0 + i$.
  – $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
Linear Hashing

- If \( N \) is 4 (the initial number of buckets), \( d_0 \) is 2 (the number of bits to represent the number 4)
- \( d_0 = 2, h_0 = h \mod 4 \)
  - Look at the last 2 bits
- \( d_1 = 2 + 1 = 3, h_1 = h \mod 16 \)
  - Look at the last 3 bits
- \( d_2 = 3 + 1 = 4, h_2 = h \mod 32 \)
  - Look at the last 4 bits
- A level counter \( L \) is kept. At level \( L \), we use only hash functions \( h_L \) and \( h_{L+1} \). Initially, \( L=0 \).
- A next counter \( \text{Next} \) is kept which indicates the next bucket to be split. Initially, \( \text{Next}=0 \).
- The number of buckets in the file at the beginning of round \( L \), \( \text{NLevel} = N \cdot 2^L \).

Linear Hashing

- For a bucket \( b \), its split image is at bucket \( b+\text{NLevel} \)
- The hash function \( h_{L+1} \) is used to distribute data entries between \( b \) and \( b+\text{NLevel} \)
- When a split occurs, \( \text{Next} \) is incremented by 1, i.e., \( \text{Next} = \text{Next} + 1 \).
- All buckets between 0 and \( \text{Next} \) are buckets that have been split
- All buckets between \( \text{Next} \) and \( \text{NLevel} \) and buckets that have yet been split
Linear Hashing - Overview

- Buckets that existed at the beginning of this round:
  - this is the range of \( h_L \)

- Buckets to be split:
  - Next

- Buckets split in this round:
  - If \( h_L (\text{search key value}) \) is in this range, must use \( h_{L+1} (\text{search key value}) \) to decide if entry is in 'split image' bucket.

- 'split image' buckets:
  - created (through splitting of other buckets) in this round

---

Linear Hashing

- **Search:** To find bucket for data entry \( r \), compute \( h_L(r) \)
- Let \( b \) denote the value of \( h_L(r) \)
  - If \( \text{Next} \leq b \leq \text{NLevel} \), then \( r \) belongs in the bucket \( b \)
  - Else, \( r \) could belong to bucket \( b \) or bucket \( b + \text{NLevel} \); in this case, we must apply \( h_{L+1}(r) \) to find out whether \( r \) belongs to the bucket \( b + \text{NLevel} \). Applying \( h_{L+1}(r) \) allows us to look at one more bit value of \( h(r) \) (the data entry's hash value) to distinguish between the bucket or its split image.
Search Example

L=0, N=4, d₀=2, d₁=3, NLevel = 4

• Search for data entry 5 where h(5)=5
  • h₀(5) = h(r) mod 4 = 5 mod 4 = 1 (i.e., it is in bucket 1). Or alternatively, 5 = 101₂ and we look at the last 2 bits
  • Next ≤ 1 ≤ NLevel. Therefore, search in bucket 1

• Search for data entry 32 where h(32) = 32
  • h₀(32) = h(32) mod 4 = 32 mod 4 = 0 (i.e., it is in bucket 0). Or alternatively, 5 = 100000₂ and we look at the last 2 bits
  • 0<Next. Therefore, we use h₁ to determine if 0 resides in bucket 0 or its split image (bucket 4)
  • h₁(32) = h(32) mod 8 = 0. Therefore 32 belongs to bucket 0.
  • If we had searched for data entry 44, h₁(44) = 4. Therefore, 44 belongs to bucket 4.
Linear Hashing - Managing change

- **Insert**: Find bucket by applying $h_L$ and possibly $h_{L+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (Maybe) Split $Next$ bucket and increment $Next$.
    - If splitting occurs at the bucket where data entry is to be inserted, then no overflow page needs to be added.
  - Directory avoided in LH by the use of overflow pages, and choosing bucket to split round-robin.

Winter 2003

Insert Example

- Insert 43*. Bucket 0 is split because Bucket 2 (where 43 is to be inserted) is full. On split, $h_{L+1}$ is used to re-distribute entries. Next is incremented. Add overflow page to hold 43*.

(This info is for illustration only!) Winter 2003
Insert Example

- Insert 37*. Bucket 1 has space for insertion.

Insert Example

- Insert 29*. Bucket 2 has no space for insertion and is the split bucket. Therefore no overflow page needs to be allocated.
Linear Hashing - Managing change

- Splitting proceeds in `rounds`: Round ends when all buckets in the range of 0 to NLevel-1 are split.
  - When Next is NLevel -1 and a split is triggered, the number of buckets after the split is twice the number at the beginning of the round. L is incremented by 1 (hash functions are switched). Next is reset to 0.
- Current round number is L
- Can choose any criterion to `trigger` split.
- Since buckets are split round-robin, long overflow chains don’t develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is implicit in how the # of bits examined is increased.

Winter 2003

Insert Example: End of a Round

- Insert 50* after 22*, 66*, and 34* have been inserted.

Winter 2003
Linear hashing

- **Deletion**: Essentially the reverse of insertion.
- If no overflow pages, equality selection is just one page I/O. In practice, about 1.2 page I/Os.

- Linear hashing vs. Extendible hashing
  - The two schemes are actually quite similar.

LH Described as a Variant of EH

- Begin with an EH index where directory has $N$ elements.
- Use overflow pages, split buckets round-robin.
- First split is at bucket 0. (Imagine directory being doubled at this point.) But elements $<1, N+1>, <2, N+2>, ...$ are the same. So, need only create directory element $N$, which differs from 0, now.
  - When bucket 1 splits, create directory element $N+1$, etc.
- So, directory can double gradually. Also, primary bucket pages are created in order. If they are *allocated* in sequence too (so that finding $i$'th is easy), we actually don’t need a directory!
Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (Duplicates may require overflow pages.)
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Duplicates handled easily.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
  - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a skewed data distribution is one in which the hash values of data entries are not uniformly distributed!

Winter 2003
Running Example

- Sailors(sid: integer, sname: string, rating: integer, age: real)
  - Assume each Sailor tuple is 50 bytes long
  - Each page can hold 80 Sailors tuples
  - We have 500 pages of such tuples
- Reserves(sid: integer, bid: integer, day: dates, rname: string)
  - Rname is the name of the person who made the reservation on behalf of the sailor
  - Assume each tuple is 40 bytes long
  - A page can hold 100 Reserves tuples
  - We have 1000 pages of such tuples
- Cost of an approach is based on the number of page I/Os

The Selection Operation - Example

SELECT *
FROM Reserves R
WHERE R.rname="Joe"

- **Approach 1**: Scan the entire relation to check for each tuple whether rname="Joe". Emit tuple if rname="Joe".
  - Reserves relation span 1000 pages
  - Cost of this approach = 1000 I/Os
- If only a few tuples have rname component equal “Joe”, this approach is clearly expensive
The Selection Operation - Example

SELECT *
FROM Reserves R
WHERE R.rname="Joe"

• **Approach 2:** If B+ tree index on rname is available, use this index to to search on rname=“Joe”
  – Cost of this approach is proportional to the height of the B+-tree (usually 3-4 page I/Os !).

Selection Condition of the form $\sigma_{R.attr \ op \ value}(R)$

• There are two dimensions to consider
  – Whether data is sorted or unsorted
  – Whether there is an index available

<table>
<thead>
<tr>
<th>Index Available</th>
<th>Use Index</th>
<th>Use Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Unavailable</td>
<td>Scan</td>
<td>Binary Search</td>
</tr>
<tr>
<td></td>
<td>Unsorted</td>
<td>Sorted</td>
</tr>
</tbody>
</table>
No Index, Unsorted Data

- Given a selection condition of the form $\sigma_{R.attr \, op \, value}(R)$ and there is no index on R.attr and data is not sorted on R.attr, solution is to scan the entire relation to look for tuples which satisfy the condition.
- Cost is the number of pages used to hold the relation.
- We have seen an example earlier.

No Index, Sorted Data

- Given a selection condition of the form $\sigma_{R.attr \, op \, value}(R)$ and there is no index on R.attr and data is sorted on R.attr,
  - perform a binary search on R.attr=value to find the first tuple where R.attr op value
- Cost is logarithmic of the number of pages used to hold the relation.
No Index, Sorted Data - Example

SELECT *
FROM Sailors S
WHERE S.rating > 5
• Recall that Sailors relation is kept in 500 pages. Assume that it is kept in ascending order of rating
• Binary search for the first tuple where rating = 5.
• Scan right until first tuple where rating is > 5. Emit current and subsequent tuples.
• Therefore, cost is $\log_2{500} + \text{number of pages which hold tuples that satisfy the condition}$ (can be 0 to 500)

With B+-Tree Index

• Given a selection condition of the form $\sigma_{R.\text{attr} \text{op} \text{value}}(R)$ and there is a B+-tree index is available on $R.\text{attr}$, the best strategy is to use the index
• Search the tree to find the first index entry that points to a qualifying tuple.
• Then scan the leaf pages of the index to retrieve all entries in which the key value satisfies the selection condition
  – For each such key value that satisfies the selection condition, retrieve the corresponding tuple
With B+-Tree Index

- If the actual tuples are stored within the leaf pages of the index, no extra cost is incurred to retrieve the corresponding tuples.

- Otherwise, extra cost is incurred to retrieve the corresponding tuples and this depends on:
  1. the number of qualifying tuples
  2. whether the index is clustered or not

- If index is clustered, cost of retrieving corresponding tuples is probably one page I/O since it is likely that all tuples are contained in one page.

- Otherwise, it is likely to be one page per corresponding tuple.
With B+-Tree Index

- Cost with a clustered B+-tree index
  - Number of I/Os for searching B+-tree (typically 3-4 pages) + Number of I/Os for searching data pages (typically 1 page)

- Cost with an unclustered B+-tree index
  - Number of I/Os for searching B+-tree (typically 3-4 pages) + Number of I/Os for searching data pages (can be 1 page per tuple)
  - Alternative is to first sort the record ids obtained from the data entries according to page ids. Then, all relevant tuples can be retrieved when the page is accessed
    - Cost is now number of I/Os for searching B+-tree (typically 3-4 pages) + Number of I/Os for searching data pages (the number of pages containing relevant tuples)

With B+-Tree Index - Example

SELECT *
FROM Reserves R
WHERE R.rname LIKE ‘C%’

- Assume that about 10,000 tuples in Reserves satisfy the above condition. Each page holds 100 tuples. Therefore, there is 100 pages of qualifying tuples.
- If B+-tree index is clustered, cost = 100 I/Os + a few more to search down the B+-tree.
- If B+-tree index is unclustered, cost is at worst 10,000 I/Os + a few more pages to search down the B+-tree (each tuple may cause a new page to be retrieved)
  - If record ids are sorted before pages are retrieved, then I/Os may be less but still more than 100 I/Os
  - Cost of scanning = 1000 I/Os (recall that we have 100,000 tuples)
  - Conclusion: If B+-tree is unclustered, it might be cheaper to simply do a file scan