Lecture 3: Hash indexing, index selection

Rasmus Pagh
Today’s lecture

• Morning session: Hashing
  – Static hashing, hash functions
  – Extendible hashing
  – Linear hashing
  – Newer techniques:
    Buffering, two-choice hashing

• Afternoon session: Index selection
  – Factors relevant for choice of indexes
  – Rules of thumb; examples and counterexamples
  – Exercises
What data in index?

- At least three possibilities:
  1) Record of key.
  2) Key and pointer to record of key.
  3) Key and list of pointers to the records containing the key (for non-unique keys).

- For simplicity, we consider the case where there is the same number of keys (B) in every disk block.
  - Case 1 with fixed length records.
  - Case 2 with fixed length keys.
Static external hashing

• Hash table:
  – Array of $N$ disk blocks. (Notation from RG.)
  – Can access block $i$ in 1 I/O, for any $i$.

• Hash function $h$:
  – Maps keys to $\{0, \ldots, N-1\}$.
  – Should be efficient to evaluate (0 I/Os).
  – Idea: $x$ is stored in block $h(x)$.

• Problem:
  – Dealing with overflows.
  – Standard solution: Overflow chains.
Problem session

• Consider the following claim from RG:

If we have \( N \) buckets, numbered 0 through \( N - 1 \), a hash function \( h \) of the form \( h(value) = (a \cdot value + b) \) works well in practice. (The bucket identified is \( h(value) \mod N \).)

• Donald Ummy uses this hash function in an application, and finds out that it performs terribly, no matter how the constants \( a \) and \( b \) are chosen.

• What might have gone wrong?
Randomized hash functions

Another approach (not mentioned in RG):

- Choose $h$ at random from some set of functions.
- This can make the hashing scheme behave well regardless of the key set.
- E.g., "universal hashing" makes chained hashing perform well (in theory and practice).
- Details out of scope for this course...
Analysis, static hashing

- Notation:
  - $N$ keys inserted,
  - Each block in the hash table can hold $B$ keys.
- Suppose that we insert $\alpha N$ keys in the hash table ("it is a fraction $\alpha$ full", "load factor $\alpha$.
- Assume $h$ is truly random.
- Expected number of overflow blocks:
  $$(1-\alpha)^{-2} \cdot 2^{-\Omega(B)} N \quad (\text{proof omitted!})$$
- Good to have many keys in each bucket (an advantage of secondary indexes).
Sometimes, life is easy

• If B is sufficiently large compared to N, all overflow blocks can be kept in internal memory.
• Lookup in 1 I/O.
• Update in 2 I/Os.
Too many overflow chains?

Can have too many overflow chains if:

- The hash function does not distribute the set of keys well ("skew").
  - Solution 1: Choose a new hash function.
  - Solution 2?: Overflow in main memory.
- The number of keys in the dictionary exceeds the capacity of the hash table.
  - Solution: Rehash to a larger hash table.
  - Better solution: ?
Doubling the hash table

• For simplicity, assume $N$ is a power of 2. Suppose $h$ is a hash function that has values of “many” (e.g. 64) bits.

• We can map a key $x$ to $\{0,\ldots,N-1\}$ by taking the log $N$ least significant bits of $h(x)$.

• Suppose that the hash table has become too small:
  – Want to double the size of the hash table.
  – Just consider one more bit of $h(x)$. 
Doubling the hash table, cont.

- Suppose \( h(x) = 0111001 \) (in binary) and the hash table has size 16.
- Then \( x \) is stored in block number 1001 (binary).
- After doubling to size 32, \( x \) should be stored in block 11001.
- Generally, all keys in block 1001 should be moved to block 01001 or 11001.
- **Conclusion**: Can rehash by scanning the table and split each block into two blocks.
Doubling, example

New key: 00100

For simplicity we assume:
• No overflow chains
• h(x) = x
Problem session

• Find some possible disadvantages of the doubling strategy. Consider:
  – Space usage vs overflows
  – System response time

• **Next**: Alternatives that address some of the disadvantages of doubling.
Linear hashing

“Virtual” blocks
- Merged with previous blocks by considering one bit less
- Turned into physical blocks as the hash table grows
Linear hashing - performance

The good:

- Resizes hash table one block at a time:
  Split a block or merge two blocks.
- Cost of resize: 3 I/Os. Cheap!

The bad:

- Increasing size of hash table may not eliminate any overflow chain.
- Uneven distribution of hash values; works best for relatively low load factors, 50-80%.
  (But variants of linear hashing improve this.)
- No worst-case guarantee on query time.
Extendible hashing

"Virtual" hash table
- no overflows

"Directory"
- mapping virtual to physical

physical hash table
Extendible hashing invariants

• Virtual hash table has no overflows - may need to increase in size.
• Physical hash table has no overflows.
• Virtual hash table is as small as possible - may need to shrink.
• "Compression": For any bit string \( s \), if we consider the virtual hash table blocks whose index ends with \( s \) then either:
  - These blocks contain more than \( B \) keys, or
  - The corresponding entries in the directory all point to the same block. (In other words, these blocks are merged.)
Extendible hashing performance

• At most 2 I/Os for every lookup.
• Only 1 I/O if directory fits in internal memory.
• Space utilization in physical hash table is 69% (expected).
• Size of directory is roughly $4N^{\frac{B}{\sqrt{N}}}$ (expected) - this is much smaller than the hash table if B is moderately large.
Buffering

• Same trick as in buffered B-trees: Don’t do updates right away, but put them in a buffer.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>1000</td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>0101</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1100</td>
<td>1010</td>
<td></td>
</tr>
</tbody>
</table>

overflow block

buffer

• Advantage: Several keys moved to the overflow block at once.
• Disadvantage: Buffer takes space.
Two-choice hashing

- **Idea:**
  - Use two hash functions, $h_1$ and $h_2$.
  - $x$ is stored in either block $h_1(x)$ or $h_2(x)$, use two I/Os for lookup.
  - When inserting $x$, choose the least loaded block among $h_1$ and $h_2$.

- Can be shown that overflow probabilities are much smaller than with one function, especially when $B$ is small.

- If two disks are available, the 2 I/Os can be done in parallel.
Today’s lecture, part 2

• Index selection
  – Factors relevant for choice of indexes
  – Rules of thumb; examples and counterexamples

• Exercises
Workload

• The workload (mix of operations to be carried out by the DBMS) has a large influence on what indexes should be created in a database.

• Other factors are:
  – the data in relations, and
  – the query plans produced by the DBMS.
Rules of thumb

• Rules of thumb can be used to guide thinking, and as a checklist.
• Are often valid in most cases, but there are always important exceptions.
• Quote from SB:

   The point of the example is that the tuner must understand the reason for the rule

• You don’t yet have the entire picture (query optimization, concurrency), but we can start reasoning about rules anyway.
Rule of thumb 1: Index the most selective attribute

• Argument: Using an index on a selective attribute will help reducing the amount of data to consider.

• Example:
  SELECT count(*) FROM R
  WHERE a>’UXS’ AND b BETWEEN 100 AND 200

• Counterexamples:
  – Full table scan may be faster than an index
  – It may not be possible/best to apply an index.
Rule of thumb 2: Cluster the most important index of a relation

• Argument:
  – Range and multipoint queries are faster.
  – Usually sparse, uses less space.

• Counterexamples:
  – May be slower on queries "covered" by a dense index.
  – If there are many updates, the cost of maintaining the clustering may be high.
  – Clustering does not help for point queries.
  – Can cluster according to several attributes by duplicating the relation!
Rule of thumb 3:
Prefer a hash index over a B-tree if point queries are more important than range queries

• Argument:
  – Hash index uses fewer I/Os per operation than a B-tree.
  – Joins, especially, can create many point queries.

• Counterexamples:
  – If a real-time guarantee is needed, hashing can be a bad choice.
  – Might be best to have both a B-tree and a hash index.
Hashing and range queries

RG page 371:

Hash-based indexing techniques cannot support range searches, unfortunately.

• **But:** they can be *used* to answer range queries in $O(1+Z/B)$ I/Os, where $Z$ is the number of results. (Alstrup, Brodal, Rauhe, 2001; Mortensen, Pagh, Patrascu 2005)

• Theoretical result on external memory (why?) - and out of scope for DBT.
Problem session

• Comparison of B-trees and extendible hashing.
  – Case 1: Directory fits internal memory.
  – Case 2: Directory on external memory.
    – Case A: B=4, N=2^{20}.
    – Case B: B=2^8, N=2^{20}.
• Consider cases 1A, 1B, 2A, 2B.
Rule of thumb 4:
Balance the increased cost of updating with the decreased cost of searching

- Argument: The savings provided by an index should be bigger than the cost.
- Counterexample:
  - If updates come when the system has excess capacity, we might be willing to work harder to have indexes at the peaks.

- If buffered B-trees are used, the cost per update of maintaining an index may be rather low. Especially if binary (!) trees are used.
Rule of thumb 5:
A non-clustering index helps when the number of rows to retrieve is smaller than the number of blocks in the relation.

• Argument: In this case it surely reduces I/O cost.
• Counterexample:
  – Even for a non-clustered index, the rows to retrieve can sometimes be found in a small fraction of the blocks (e.g. salary, clustered on date of employment).
Rule of thumb 6: Avoid indexing of small tables.

• Argument: Small tables can be kept in internal memory, or read entirely in 1 or 2 I/Os.

• Counterexample:
  – If the index is in main memory, it might still give a speedup.
Conclusion

• Indexing is a complicated business!
• Understanding the various index types and their performance characteristics, as well as the characteristics of the database at hand and its workload allows informed indexing decisions.
• Rules of thumb can be used to guide thinking.
• More complications to come!
Tip: Clustered indexing in Oracle

- Default in Oracle is to store tuples in a heap (think insertion order).
- Is clustered according to the primary key, if "ORGANIZATION INDEX" is added after the schema when creating the relation.
- To cluster according to a non-unique attribute A, declare a composite primary key (A,P), where P is a unique key.
Exercises

Hand-outs: