Lecture 7
Hash-based Indexing
Chapter 11: Ramakrishnan & Gehrke
Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl446
Lecture Outline
Hash Indexes (Ευρετήρια Κατακερματισμού)

- 11.1) Static Hashing (Στατικός Κατακερματισμός)
- Dynamic Hashing (Δυναμικός Κατακερματισμός)
  - 11.2) Extendible Hashing (Επεκτατός Κατακερματισμός)
  - 11.3) Linear Hashing (Γραμμικός Κατακερματισμός)
  - 11.4) Extendible vs Linear Hashing
Introductory Remarks
(Εισαγωγικές Επισημάνσεις)

• As for any index, 3 alternatives for data entries k*:
  – Alternative 1: <k>
  – Alternative 2: <k, RID>
  – Alternative 3: <k, [RID₁, RID₂, …, RIDₙ]>
  – Choice orthogonal to the indexing technique

• Hashing (Κατακερματισμός): key-to-address transformation: involves computing the address of a data item by computing a function on the search key value.

• Hash Indexes (Ευρετήρια Κατακερματισμοί) are best for equality queries (Επερωτήσεις Ισότητας). Cannot support range queries.
Hash Function \( h(k) \)  
(Συνάρτηση Κατακερματισμού)

- **Hash function \([h(key)]\):** Maps the key to a bucket (κάδο) where the key is expected to belong.

- A good hash function has the following properties:
  - **Distributes keys uniformly** (ομοιόμορφα) - all buckets are equally likely to be picked and at random (τυχαία) - similar hash keys should be hashed to very different buckets.
  - **Low Cost.** Plain hash functions (rather than cryptographic hash functions such as MD5, SHA1) usually have a low computational cost.
  - **Determinism:** for a given input value it always generates the same hash value.

- We shall utilize a **Trivial Hash Function** (τετριμμένη συνάρτηση κατακερματισμού), i.e., the data itself (interpreted as an integer in binary notation). E.g., \( 44_{10} = 101100_2 \)

- Which Bucket does key \( k \) belong to: \( h(k) \mod N \) (\( N = \# \) of buckets). If \( N \) power of two (i.e., \( N=2^d \), \( d \in \mathbb{N} \)), then these are the \( d \) least significant bits.

\[
\begin{array}{c|c|c}
\text{key} & h(\text{key}) & \% \ N \\
\hline
0 & 1 & \\
\hline
\end{array}
\]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( h(k) \mod 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>101100</td>
<td>2</td>
</tr>
<tr>
<td>000011</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ e.g., \text{How to obtain last 2-bits?} \]

\[ \text{Bitwise Modulo: } x \mod 4 = x \& 3 \]

\[
\begin{array}{c|c|c}
\text{Bit} & \text{Bitwise Modulo} \\
\hline
0 & 0 \\
1 & 3 \\
2 & 2 \\
3 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\hline
\text{Bit} & \text{Bitwise Modulo} & \text{Bitwise Modulo} & \text{Bitwise Modulo} \\
\hline
0 & 000011 & 000011 & 000011 \\
1 & 000010 & 000010 & 000010 \\
2 & 000000 & 000000 & 000000 \\
\hline
\end{array}
\]
Static Hashing
(Στατικούς Κατακερματισμούς)

- Build a **fixed structure** at index construction time.
- Data Entries are stored on a number of **successive primary pages** (πρωτοβάθμιες σελίδες).
  - Primary pages are **fixed, allocated sequentially** during index construction. **Overflow pages** (σελίδες υπερχείλισης) are utilized when primary pages get full.
  - **Primary** Pages are never **de-allocated** during deletions.
  - That is similar to the way ISAM indexes are constructed...

\[ k \xrightarrow{h} h(k) \mod N \]

\[ \begin{array}{c}
  0 \\
  1 \\
  \vdots \\
  N-1 \\
\end{array} \]

- **Primary bucket pages**
- **Overflow pages**

k: Search Key , e.g., age field
h(k): Identifies bucket for data entry k*
Static Hashing
(Στατικός Κατακερματισμός)

- **Search**: Ideally 1 I/O (unless record is located in overflow chain). **Insert/Delete**: 2 I/Os (read and write) page.
- **Drawback**: Long overflow chains (Αλυσίδες Σελίδων Υπερχείλισης) can develop and degrade performance.

**How to avoid overflow chains?**

1. **80% Occupancy**: By initially keeping pages 80% full we can avoid overflow pages if the file does not grow too much.
2. **Rehashing (Επανακαταρτισμός)**: Hash the file with a different hash function (see next slide) to achieve 80% occupancy and no overflows. Drawback: Takes time (we need to rehash the complete DB)!
3. **Dynamic Hashing**: Allow the hash function to be modified dynamically to accommodate the growth/shrink of the database (i.e., essentially rehash selected, rather than all, items)
   - Extendible Hashing (Επεκτακτό Κατακερματισμό)
   - Linear Hashing (Γραμμικό Κατακερματισμό)
To understand the motivation of Extendible Hashing consider the following situation:

A Bucket (primary page) becomes full (e.g., page 00 on left). Why not re-organize file by doubling # of buckets?

Answer: The entire file has to be read once and written back to disk twice to achieve the reorganization, which is expensive!
Extendible Hashing (Επεκτατός Κατακερματισμός)

- **Basic Idea:** Use directory of pointers to buckets and **double the directory** instead of Doubling the Data file.
  - Directory much smaller than file, so doubling is much cheaper.
- Just split the bucket that overflowed NOT ALL of them
  - Only one page of data entries is split.
  - Additionally, no overflow pages are constructed!

![Diagram showing Extendible Hashing concept](image-url)
Extendible Hashing: Search

(Επεκτατός Κατακερματισμός: Αναζήτηση)

- **Example:** Locate data entry \( r \) with hash value \( h(r) = 5 \) (binary 101). Look at directory element 01 (i.e., “Global-depth least-significant bits of \( h(r) \), δηλ., 2 λιγότερα σημαντικά ψηφία”).

- We then follow the pointer to the data page (bucket B in figure).
Extendible Hashing: **Insert**

(Eπεκτατός Κατακερματισμός: Εισαγωγή)

**Insert Algorithm Outline**

- Find target buffer: Done similarly to Search
- If target bucket is **NOT full**, insert and finish (e.g., insert \( h(r) = 9 \), which is binary 1001, can be inserted to bucket B).
- If target bucket is full, **split** it (allocate new page and re-distribute). E.g., insertion of \( h(r) = 20 \) (10100) causes the split of bucket A and redistribution between A and A2.

Insert 20

Insertion of \( h(r) = 20 \) (10100) causes the split of bucket A and redistribution between A and A2.

Bucket A2 needs to be pointed from the directory. Thus, we will double the directory (next slide).
• When does bucket split cause directory doubling?
  • When target bucket is full AND Local Depth == Global Depth
    • Otherwise, a red pointer is available (i.e., vacant page is already avail.).
• Notice that after doubling some pointers (red) are redundant (those will be utilized in subsequent inserts).
Comments on Extendible Hashing
(Σχόλια για τον Επεκτατό Κατακερματισμό)

- **Global depth of directory**: Tells us how many least significant bits to utilize during the selection of the target bucket.
  - Initially equal to $\log_2(#\text{Buckets})$, e.g., $\log_28=3$
  - Directory Doubles => Increment Global Depth
- **Local depth of a bucket**: Tells as how many least significant bits to utilize to determine if an entry belongs to a given bucket.
  - Bucket is Split => Increment Local Depth
- **(GlobalDepth – LocalDepth)** can be larger than 1 (e.g., if corresponding buckets are continuously splitted leaving in that way the local depth of other nodes small while global depth increases)

During an Insertion if Local Depth == Global Depth then we need to split!
Extendible Hashing: **Delete**

(Επεκτατός Κατακερματισμός: **Διαγραφή**)

- **Delete**: Essentially the **reverse** operation of **insertion**
- If removal of data entry makes **bucket empty** then **merge** with `split image` (e.g., delete 32,16, then merge with A2)
- If **every bucket** is pointed by two directory elements we should halve the directory (although not necessary for correctness)
Comments on Extendible Hashing
(Σχόλια για τον Επεκτατό Κατακερματισμό)

• **Equality Search Cost:** If directory fits in memory then answered with 1 disk access; else 2.
  – **Static Hashing** on the other hand performs equality searches with 1 I/O (assuming no collisions).

• Yet, the Extendible Hashing Directory can usually easily fit in main memory, thus same cost.

**Other issues:**

• Directory can grow large if the distribution of hash values is skewed (ασύμμετρη κατανομή) (e.g., some buckets are utilized by many keys, while others remain empty).

• Multiple entries with same hash value (collisions) cause problems … as splitting will not redistribute equally the keys.
Linear Hashing (LH)  
(Γραμμικός Κατακερματισμός - ΓΚ)

• Another **dynamic hashing** scheme (like EH).
• LH handles the problem of long overflow chains (presented in Static Hashing) **without using a directory** (what EH does)

**Idea:** Use a family of hash functions $h_0, h_1, h_2, \ldots$ where each hash function maps the elements to twice the range of its predecessor, i.e.,

- if $h_i(r)$ maps a data entry $r$ into $M$ buckets, then $h_{i+1}(r)$ maps a data entry into one of $2M$ buckets. Hash functions are like below:
  - $h_i(key) = h(key) \mod(2^iN)$, $i=0,1,2\ldots$ and $N=\text{"initial-#-of-buckets"}$
- We proceed in **rounds** of splits: During round $Level$ only $h_{Level}(r)$ and $h_{Level+1}(r)$ are in use.
- The buckets in the file are **split** (every time we have an overflow), **one-by-one** from the **first** to the **last** bucket, thereby **doubling the number of buckets**.
Insert Algorithm Outline:

- Find target buffer (similarly to search with $h_{level}(r)$ and $h_{level+1}(r)$)
- If target bucket is **NOT full**, insert and finish (e.g., insert $h(r)=9$, which is binary $1001$, can be inserted to bucket B).
- If target bucket is full:
  - Add overflow page and insert data entry. (e.g., by inserting $h(r)=43$ (101011) causes the split of bucket A and redistribution between A and A2)
  - **Split Next** bucket and **increment Next** (can be performed in batch mode)
    - Note that $32(100000)$, $44(101100)$, $36(100100)$

![Diagram showing linear hashing process](image-url)
Linear Hashing: Insertion Remarks
(Γραμμικός Κατακερματισμός: Επισημάνσεις Εισαγωγής)

- The buckets in the file are split (every time we have an overflow), one-by-one from the first to the last bucket $N_R$ (using Next index), thereby doubling the number of buckets.
- Since buckets are split round-robin, long overflow chains don’t develop (like static hashing) as eventually every bucket is split!
- LH can choose any criterion to `trigger’ (προκαλέσει) split:
  - e.g., Split whenever an overflow page is added.
  - e.g., Split whenever the index is e.g., 75% full.
  - Many other heuristics could be utilized.
Linear Hashing: Increasing Level after Insert

If \( \text{Next} = N_R \) (after overflow) then level is increased by 1 (thus h2, h1 will be utilized) and Next becomes 0

- Below the addition of 50* (110010) causes Next to become equal to 4 thus the Level is increased by one.

\[
\begin{array}{c|c|c}
\text{Level}=0 & \text{PRIMARY PAGES} & \text{OVERFLOW PAGES} \\
\hline
\h_1 & h_0 & \h_1 & h_0 \\
\hline
000 & 00 & 32* & 000 & 00 \\
001 & 01 & 9* 25* & 001 & 01 \\
010 & 10 & 66*18*10* 34* & 010 & 10 \\
011 & 11 & 31*35* 7* 11* & 011 & 11 \\
100 & 00 & 44*36* & 100 & 00 \\
101 & 01 & 5* 37*29* & 101 & 11 \\
110 & 10 & 14*30*22* & 110 & 11 \\
\hline
\end{array}
\]

Insert 50

\[
\begin{array}{c|c|c}
\text{Level}=1 & \text{PRIMARY PAGES} & \text{OVERFLOW PAGES} \\
\hline
\text{Next}=0 & 32* & 50* \\
\hline
000 & 00 & 9* 25* \\
010 & 10 & 66*18*10* 34* \\
011 & 11 & 43* 35* 11* \\
100 & 00 & 44* 36* \\
101 & 11 & 5* 37* 29* \\
110 & 11 & 31*7* \\
\hline
\end{array}
\]

Corresponding Bucket (Κάδοι Αντιστοιχίας)
Overview of LH File
(Ανασκόπηση Αρχείου ΓΚ)

- Assume that we are in the middle of an execution.
- Then the Linear Hash file has the following structure

Splitted Pages

To insert: Utilize $h_{\text{Level}} \parallel h_{\text{Level+1}}$

Unsplitted Pages

To insert: Utilize $h_{\text{Level}}$

New Pages from Split

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

Buckets that existed at the beginning of this round:

Next

$N_R$
Linear Hashing: Search

(Search: To find bucket for data entry \( r \), find \( h_{\text{Level}}(r) \):

- **Unsplit Bucket:** If \( h_{\text{Level}}(r) \) in range \([\text{Next}..\text{N}_R]\) then \( r \) belongs here (e.g., 9)
- **Split Bucket:** If \( h_{\text{Level}}(r) \) maps to bucket smaller than Next (\( i.e., a \) bucket that was split previously, then \( r \) could belong to bucket \( h_{\text{Level}}(r) \) or bucket \( h_{\text{Level}}(r) + \text{N}_R \); must apply \( h_{\text{Level}+1}(r) \) to find out (e.g., \( 44_{10} = 101100_2 \)))
• Hash-based indexes: best for equality searches, cannot support range searches.
• Static Hashing can lead to long overflow chains.
• Extendible Hashing uses a Directory which keeps track of new buckets and which doubles periodically.
  – NO overflow pages: A full bucket is split when a new data entry is to be added to it. (however, Duplicates may require overflow pages)
  – Can get large with skewed data (e.g., all entries fall into a few buckets); while in Linear Hashing each page will eventually be split by the round robin algorithm
    • Additional I/O if this does not fit in main memory.
Chapter Summary
(Σύνοψη Κεφαλαίου)

- **Linear Hashing** avoids **directory** by splitting buckets **round-robin**, and using **overflow pages**.
  - Overflow pages not likely to be long (1-2 pages).
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
    - Can tune criterion for triggering splits (e.g., index size, etc)

- Linear and Extendible Hashing are quite similar to each other but different.