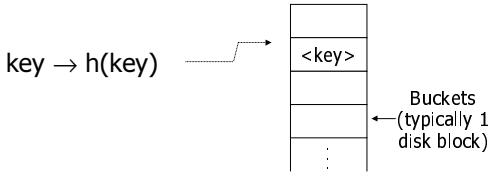
CS 245: Database System Principles

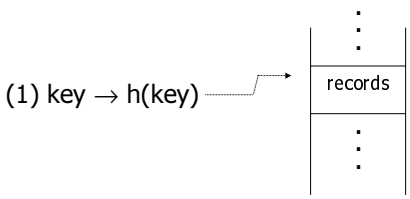
Notes 5: Hashing and More

Hector Garcia-Molina
(Some modifications by Chris Olston)

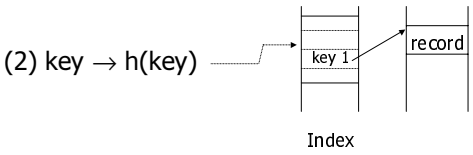
Hashing



Two alternatives



Two alternatives



- Alt (2) for "secondary" search key

Example hash function

- Key = 'x_1 x_2 ... x_n' *n* byte character string
- Have *b* buckets
- h: add $x_1 + x_2 + ... x_n$
  - compute sum modulo *b*

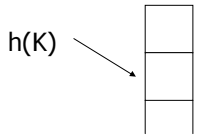
- This may not be best function …
- Read Knuth Vol. 3 if you really need to select a good function.

Good hash function: ☞ Expected number of keys/bucket is the same for all buckets

Within a bucket:

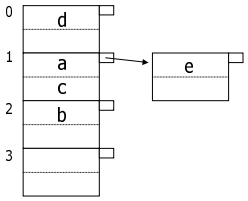
- Do we keep keys sorted?
- Yes, if CPU time critical
  & Inserts/Deletes not too frequent

Next: example to illustrate
inserts, overflows, deletes



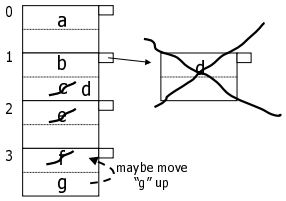
EXAMPLE 2 records/bucket

INSERT:
h(a) = 1
h(b) = 2
h(c) = 1
h(d) = 0
h(e) = 1



EXAMPLE: deletion

Delete:
e
f
c



Rule of thumb:

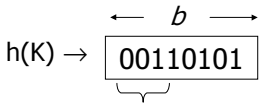
- Try to keep space utilization
  between 50% and 80%
  Utilization = # keys used / total # keys that fit
- If < 50%, wasting space
- If > 80%, overflows significant
  depends on how good hash function is & on # keys/bucket

How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear

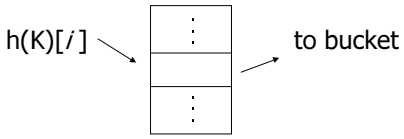
Extensible hashing: two ideas

(a) Use i of b bits output by hash function

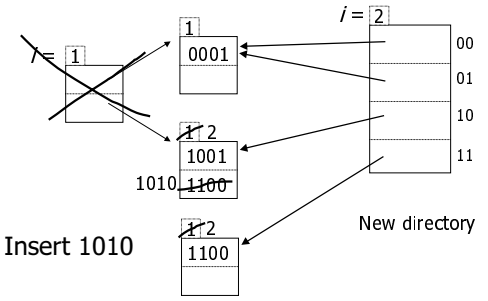


use $i \rightarrow$ grows over time….

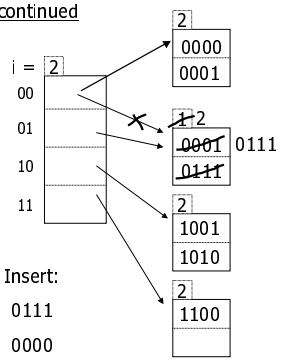
(b) Use directory



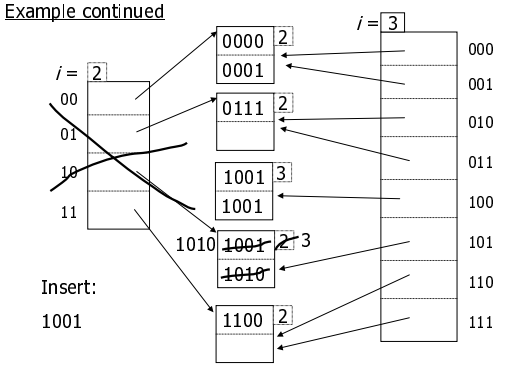
Example: h(k) is 4 bits; 2 keys/bucket



Example continued



Example continued



Extensible hashing: deletion

- No merging of blocks
- Merge blocks
  and cut directory if possible
  (Reverse insert procedure)

Deletion example:

- Run thru insert example in reverse!

Summary Extensible hashing

⊕ Can handle growing files
- with less wasted space
- with no full reorganizations

⊖ Indirection
(Not bad if directory in memory)

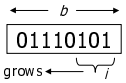
⊖ Directory doubles in size
(Now it fits, now it does not)

Linear hashing

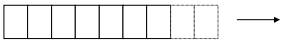
- Another dynamic hashing scheme

Two ideas:

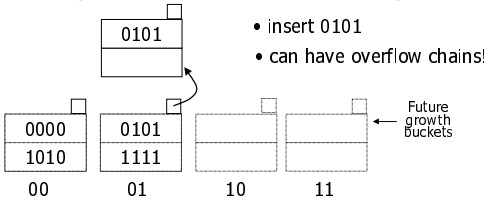
(a) Use *i* low order bits of hash



(b) File grows linearly



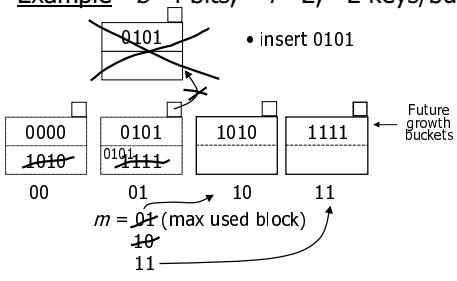
Example *b*=4 bits, *i* =2, 2 keys/bucket



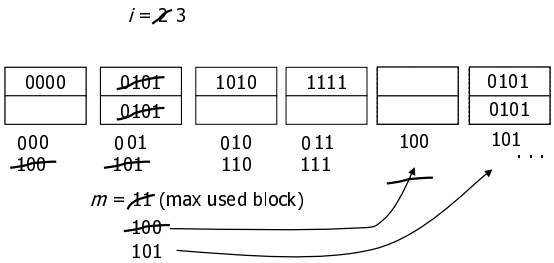
m = 01 (max used block)

Rule If h(k)[*i*] ≤ *m*, then
look at bucket h(k)[*i*]
else, look at bucket h(k)[*i*] - 2^{i-1}

Example *b*=4 bits, *i* =2, 2 keys/bucket



Example Continued: How to grow beyond this?



☛ When do we expand file?

- Keep track of: $\frac{\text{\# used slots}}{\text{total \# of slots}} = U$
- If U > threshold then increase *m* (and maybe *i*)

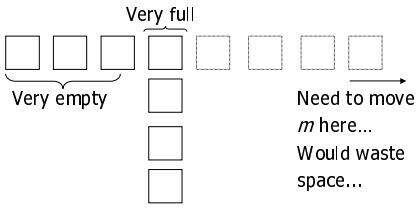
Summary Linear Hashing

⊕ Can handle growing files
- with less wasted space
- with no full reorganizations

⊕ No indirection like extensible hashing

⊖ Can still have overflow chains

Example: BAD CASE



Summary

Hashing
- How it works
- Dynamic hashing
  - Extensible
  - Linear

Next:

- Ordered Indexing vs Hash Indexing
- Index definition in SQL
- Multiple key access
- Index selection problem

Ordered Indexing vs Hash Indexing

- Hashing good for probes given key

e.g.,
```
SELECT …
FROM R
WHERE R.A = 5
```

Ordered Indexing vs Hash Indexing

- ORDERED INDEXING (Including B-Trees) good for

  Range Searches:

  e.g., SELECT
  FROM R
  WHERE R.A > 5

Index definition in SQL

- Create index name on rel (attr)
- Create unique index name on rel (attr)
  → defines candidate key
- Drop INDEX name

Note CANNOT SPECIFY TYPE OF INDEX
(e.g. B-tree, Hashing, …)
OR PARAMETERS
(e.g. Load Factor, Size of Hash,...)

... at least in SQL...

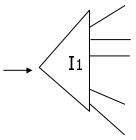
Note ATTRIBUTE LIST ⇒ MULTIKEY INDEX
(next)
e.g., CREATE INDEX foo ON R(A,B,C)

Multi-key Index

Motivation: Find records where
DEPT = "Toy" AND SAL > 50k

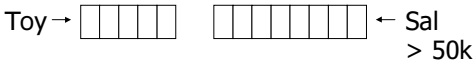
Strategy I:

- Use one index, say Dept.
- Get all Dept = "Toy" records
  and check their salary



Strategy II:

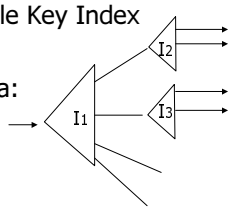
- Use 2 Indexes; Manipulate Pointers



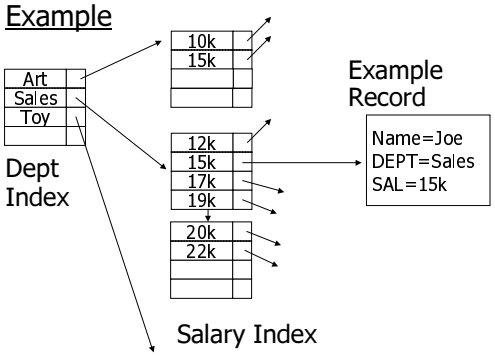
Strategy III:

- Multiple Key Index

One idea:

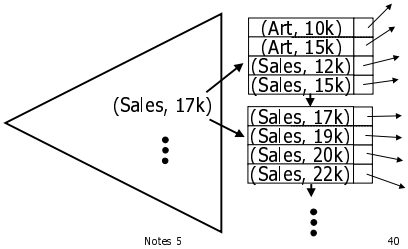


Example



Efficient B-Tree Implementation

Index entries are pairs (dept, salary)
Relies on total ordering over pairs:

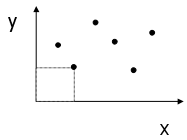


For which queries is this index good?

- ☐ Find RECs Dept = "Sales" ∧ SAL=20k
- ☐ Find RECs Dept = "Sales" ∧ SAL ≥ 20k
- ☐ Find RECs Dept = "Sales"
- ☐ Find RECs SAL = 20k

Interesting application:

- Geographic ("spatial") Data

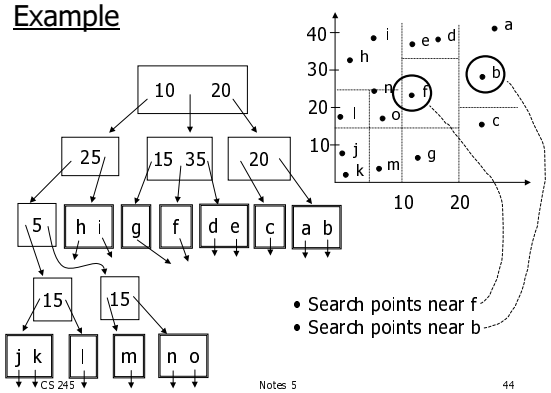


DATA:
<X_1,Y_1, Attributes>
<X_2,Y_2, Attributes>
⋮

Queries:

- What city is at <Xi,Yi>?
- What is within 5 miles from <Xi,Yi>?
- Which is closest point to <Xi,Yi>?

Example



Queries

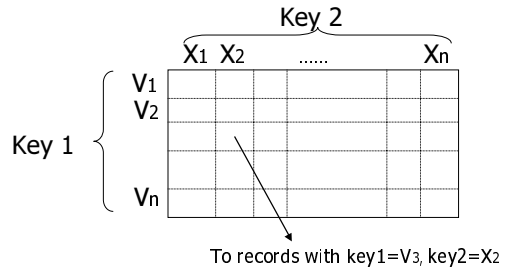
- Find points with Yi > 20
- Find points with Xi < 5
- Find points "close" to i = <12,38>
- Find points "close" to b = <7,24>

- Many types of spatial index structures have been suggested
  - Quad Trees
  - R Trees
  - K-D Trees
  - …

Three more types of multi key indexes

- Grid
- R-Tree
- Partitioned hash

Grid Index

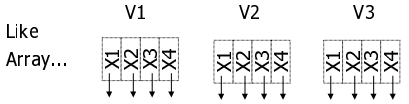


CLAIM

- Can quickly find records with
  - key 1 = V_i ∧ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j
- And also ranges….
  - E.g., key 1 ≥ V_i ∧ key 2 < X_j

☛ But there is a catch with Grid Indexes!

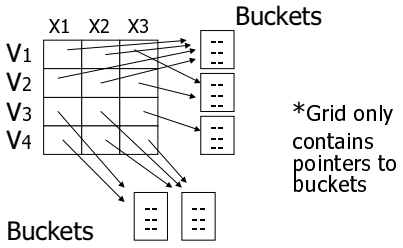
- How is Grid Index stored on disk?



Problem:

- Need regularity so we can compute position of <Vi,Xj> entry

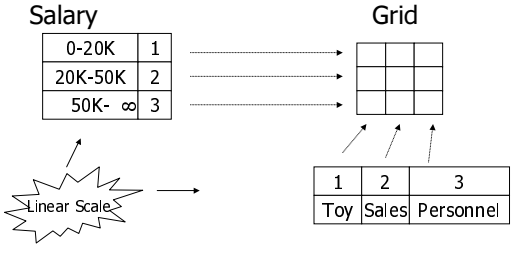
Solution: Use Indirection



With indirection:

- Grid can be regular without wasting space
- We do have price of indirection

Can also index grid on value ranges



Grid files

⊕ Good for multiple-key search

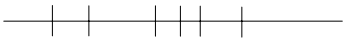
⊖ Space, management overhead
(nothing is free)

⊖ Need partitioning ranges that evenly split keys

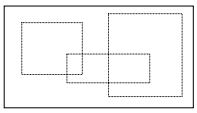
R-Trees

Extension of B-Tree idea

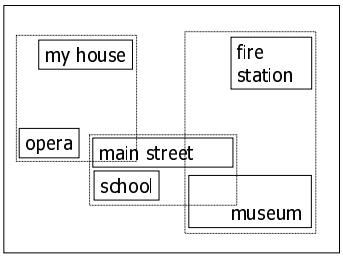
B-Tree: partition 1-D space into line segments



R-Tree: partition n-D space into regions (typically rectangles)



Example R-Tree



R-Tree vs. B-Tree

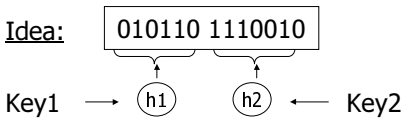
- Multi-key access
- Partitions not disjoint (may overlap)
  → May have to search multiple branches!
- Try to choose partitions to minimize amount of overlap
  - When initially build index
  - When inserting new keys

R-Tree vs. Grid files

⊕ Dynamically selects/adjusts partitions
⊕ No storage for empty regions
⊖ Overlap: search multiple partitions

Partitioned hash function

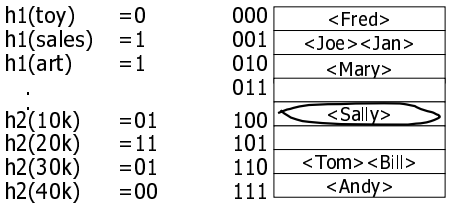
Idea: 010110 1110010



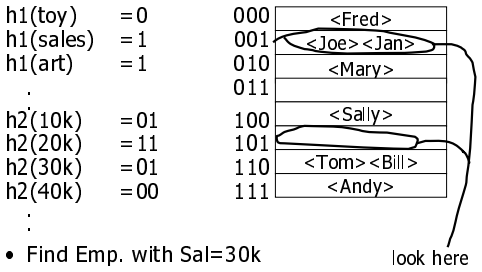
EX:

h1(toy)	= 0	000
h1(sales)	= 1	001 <Fred>
h1(art)	= 1	010
.		011
h2(10k)	= 01	100
h2(20k)	= 11	101 <Joe><Sally>
h2(30k)	= 01	110
h2(40k)	= 00	111
:		

Insert <Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

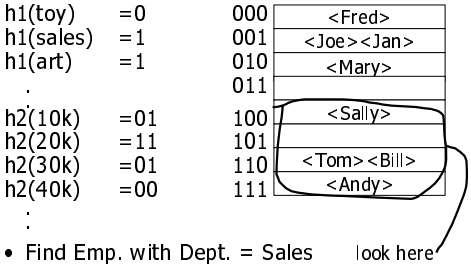


- Find Emp. with Dept. = Sales ∧ Sal=40k



- Find Emp. with Sal=30k

look here



- Find Emp. with Dept. = Sales

look here

Index Selection

- Problem: many indexing options, how to choose?
  - Primary index column (sequencing column)
  - Secondary index columns
  - Dense/sparse
  - Index types
    - B-trees, hashing, multi-key, and more…

Index Selection in Practice

- Until recently, most DBMS's only supported B-trees (some multi-column)
  - Hash indexes rare
- New generation of DBMS's:
  - Focus on domain-specific indexes
    - E.g., images, text, spatial data, multimedia

The best index selection depends on the query workload :

- Reads vs. writes
- Which columns accessed
- Selection predicates: equality, range, …
- Multiple selection predicates common?
- Joins vs. single-table queries
- Etc…

Automatic Index Selection

- Problem –
  - Given: database, workload
  - Find: optimal set of indexes to build
- Can't the computer do this for us?
- Current research topic:
  - E.g., [Chaudhuri and Narasayya, VLDB'97]

Data Independence

Nice feature of modern DBMS's: Physical Data Independence

- User doesn't need to be aware of physical design (i.e., don't have to rewrite queries)

- (Also: Logical Data Independence
  - Logical view facilities shield users from changes to schema

)

In the old days …

- "Pre-relational" or "legacy" systems
  - No physical data independence
  - The application designer wrote the data access code
    - e.g., scan table, traverse index, …
  - Add/remove an index → have to change the code ☹
  - Data distribution changes → access method in use may not be the most efficient ☹

Relational Database "Revolution"

- Physical data independence achieved via declarative query language interface
  - Say what you want, not how to get it
  - Rely on query optimizer to select a fast access method
    - Query optimization covered later in this class…

Summary

Post hashing discussion:

- Ordered Indexing vs. Hash Indexing
- SQL Index Definition
- Multiple Key Access
  - Multi Key Index
    Variations: Grid, Geo Data
  - Partitioned Hash
- Index Selection Problem
- Physical Data Independence

Reading Chapter 5

- Skim the following sections:
  - 5.3.3, 5.3.4, 5.3.5, 5.3.6
  - 5.4.2, 5.4.3, 5.4.4
- Read the rest

The BIG picture….

- Chapters 2 & 3: Storage, records, blocks…
- Chapter 4 & 5: Access Mechanisms
  - Indexes
  - B trees
  - Hashing
  - Multi key
- Chapter 6 & 7: Query Processing