Data Storage & Indexes

Instructor: Matei Zaharia

cs245.stanford.edu
Outline

Co-designing storage and compute (paper)

Indexes
Outline

Co-designing storage and compute (paper)

Indexes
C-Store Storage

The storage construct was a “projection”; what does that mean?
C-Store Compression

Five types of compression:
» Null suppression
» Dictionary encoding
» Run-length encoding
» Bit-vector encoding
» Lempel-Ziv

Tradeoff: size vs ease of computation
API for Compressed Blocks

<table>
<thead>
<tr>
<th>Properties</th>
<th>Iterator</th>
<th>Access</th>
<th>Block Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>isOneValue()</td>
<td>getNext()</td>
<td></td>
<td>getSize()</td>
</tr>
<tr>
<td>isValueSorted()</td>
<td></td>
<td>asArray()</td>
<td>getStartValue()</td>
</tr>
<tr>
<td>isPosContig()</td>
<td></td>
<td></td>
<td>getEndPosition()</td>
</tr>
</tbody>
</table>

Table 1: Compressed Block API

<table>
<thead>
<tr>
<th>Encoding Type</th>
<th>Sorted?</th>
<th>1 value?</th>
<th>Pos. contig.?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLE</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Bit-string</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Null Supp.</td>
<td>no/yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Lempel-Ziv</td>
<td>no/yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Dictionary</td>
<td>no/yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Uncompressed</td>
<td>no/yes</td>
<td>no</td>
<td>no/yes</td>
</tr>
</tbody>
</table>
Using the Block API

\[
\text{COUNT(COLUMN } \, c1) \\
b = \text{GET NEXT COMPRESSED BLOCK FROM } \, c1 \\
\text{WHILE } b \text{ IS NOT NULL} \\
\text{IF } b \text{.ISONEVALUE()} \\
\quad x = \text{FETCH CURRENT COUNT FOR } b \text{.GETSTARTVAL()} \\
\quad x = x + b \text{.GETSIZE()} \\
\text{ELSE} \\
\quad a = b \text{.ASARRAY()} \\
\quad \text{FOR EACH ELEMENT } i \text{ IN } a \\
\quad \quad x = \text{FETCH CURRENT COUNT FOR } i \\
\quad x = x + 1 \\
\quad b = \text{GET NEXT COMPRESSED BLOCK FROM } \, c1
\]

Figure 2: Pseudocode for Simple Count Aggregation
Data Size with Each Scheme

![Graph showing data size with each scheme](image)

(a) Runs of length 50
(b) Runs of length 1000

**Figure 4:** Compressed column sizes for varied compression schemes on column with sorted runs of size 50 (a) and 1000 (b)
Performance with Each Scheme

(a) Runs of length 50
(b) Runs of length 1000

How would the results change on SSDs?
Outline

Co-designing storage and compute (paper)

Indexes
Key Operations on an Index

Find all records with a given value for a key
  » Key can be one field or a tuple of fields (e.g. country=“US” AND state=“CA”)
  » In some cases, only one matching record

Find all records with key in a given range

Find nearest neighbor to a data point?
Tradeoffs in Indexing

- Improved query performance
- Cost to update indexes
- Size of indexes
Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing

Many standard data structures, but adapted to work well on disk
## Sequential File

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
Sparse 2nd level

10  90  170  250

330  410  490  570

Sequential File

10  30  50  70

90  110  130  150

170  190  210  230

10  20

30  40

50  60

70  80

90  100

Note: file, index blocks may or may not be contiguous on disk
Sparse vs Dense Tradeoff

**Sparse:** Less space usage, can keep more of index in memory

**Dense:** Can tell whether any record exists without accessing file

(Later: sparse better for insertions, dense needed for secondary indexes)
Terms

Search key of an index
Primary index (on primary key of ordered files)
Secondary index
Dense index (contains all search key values)
Sparse index
Multi-level index
Handling Duplicate Keys

For a primary index, can point to first instance of each item (assuming blocks are linked)

For a secondary index, need to point to a list of records since they can be anywhere
Deletion: Sparse Index

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>70</td>
<td>50</td>
</tr>
<tr>
<td>90</td>
<td>60</td>
</tr>
<tr>
<td>110</td>
<td>70</td>
</tr>
<tr>
<td>130</td>
<td>80</td>
</tr>
<tr>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>
Deletion: Sparse Index

– delete record 40

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>30</td>
</tr>
<tr>
<td>110</td>
<td>40</td>
</tr>
<tr>
<td>130</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>80</td>
</tr>
</tbody>
</table>
Deletion: Sparse Index
– delete record 40
Deletion: Sparse Index

– delete record 30

\(\begin{array}{c}
10 \\
30 \\
50 \\
70 \\
90 \\
110 \\
130 \\
150 \\
\end{array}\) \rightarrow \(\begin{array}{c}
10 \\
20 \\
30 \\
40 \\
30 \\
40 \\
50 \\
60 \\
70 \\
80 \\
\end{array}\)
Deletion: Sparse Index

– delete record 30

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>90</th>
<th>110</th>
<th>130</th>
<th>150</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
</tr>
</tbody>
</table>
Deletion: Sparse Index
– delete records 30 & 40
Deletion: Sparse Index

– delete records 30 & 40
Deletion: Sparse Index
– delete records 30 & 40
Deletion: Dense Index

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>70</td>
<td>70</td>
<td>80</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>80</td>
</tr>
</tbody>
</table>
Deletion: Dense Index
– delete record 30
Deletion: Dense Index
– delete record 30
Deletion: Dense Index

– delete record 30
### Insertion: Sparse Index

- insert record 34

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>60</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>60</td>
</tr>
</tbody>
</table>
Insertion: Sparse Index

– insert record 34
Insertion: Sparse Index

- insert record 34

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>60</td>
<td>34</td>
</tr>
</tbody>
</table>

- our lucky day!
  we have free space
  where we need it!
Insertion: Sparse Index

– insert record 15

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>30</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>60</th>
</tr>
</thead>
</table>
Insertion: Sparse Index

– insert record 15

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>20</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Insertion: Sparse Index

- **insert record 15**

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

- **Illustrated:** Immediate reorganization
- **Variation:**
  - insert new block (chained file)
  - update index
Insertion: Sparse Index

– insert record 25
Insertion: Sparse Index

– insert record 25

overflow blocks (reorganize later...)
Secondary Indexes

<table>
<thead>
<tr>
<th>Ordering field</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>60</td>
</tr>
</tbody>
</table>
Secondary Indexes

Sparse index:

<table>
<thead>
<tr>
<th></th>
<th>30</th>
<th>20</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ordering field:

<table>
<thead>
<tr>
<th></th>
<th>30</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Secondary Indexes

Sparse index:

30
20
80
100

30
50
20
70
80
40
100
10
90
60

does not make sense!
Secondary Indexes

Dense index:

Ordering field

<table>
<thead>
<tr>
<th>10</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
Secondary Indexes

Dense index:

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sparse higher level

Ordering field

<table>
<thead>
<tr>
<th>30</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>70</td>
</tr>
<tr>
<td>80</td>
<td>40</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>90</td>
<td>60</td>
</tr>
</tbody>
</table>
With Secondary Indexes

Lowest level is dense

Other levels are sparse

Pointers are record pointers (not block)
Duplicate Values in Secondary Indexes
Another Benefit of Buckets

Can compute complex queries through Boolean operations on record pointer lists

Consider an employee table with foreign keys for department and floor:

<table>
<thead>
<tr>
<th>DeptID</th>
<th>...</th>
<th>EmpID</th>
<th>Name</th>
<th>DeptID</th>
<th>FloorID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>Alice</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>Bob</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FloorID</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Query: Get Employees in (Toy Dept) AND (2\textsuperscript{nd} floor)

Intersect “Toy” bucket and “2\textsuperscript{nd} floor” buckets to get list of matching employees
This Idea is Used in Text Information Retrieval

Documents

...the cat is fat ...

...was raining cats and dogs...

...Fido the dog ...
This Idea is Used in Text Information Retrieval

Inverted lists

Documents

...the cat is fat ...

...was raining cats and dogs...

...Fido the dog ...
Common Technique: More Info in Index Entries

Answer queries like “cat within 5 words of dog”
Conventional Indexes

Pros:
- Simple
- Index is sequential file (good for scans)

Cons:
- Inserts expensive, and/or
- Lose sequentiality & balance
Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing
B-Trees

Another type of index
  » Give up on sequentiality of index
  » Try to get “balance”

Note: the exact data structure we’ll look at is a B+ tree, but plain old “B-trees” are similar
B+ Tree Example

Root

3 5 11

30 35

100 101 110

120 130

150 156 157 179

180 200

(n = 3)
Sample Non-Leaf

- 57
- 81
- 95

< 57  
57 \leq k < 81  
81 \leq k < 95  
\geq 95

To keys
Sample Leaf Node

From non-leaf node

57  81  95

To record with key 57
To record with key 81
To record with key 95

to next leaf in sequence
Size of Nodes on Disk

\[
\begin{align*}
\{ & n + 1 \text{ pointers} \\
\{ & n \text{ keys} \\
\end{align*}
\]

(Fixed size nodes)
Don’t Want Nodes to be Too Empty

Use at least

Non-leaf: $\left\lceil (n+1)/2 \right\rceil$ pointers

Leaf: $\left\lfloor (n+1)/2 \right\rfloor$ pointers to data
Example: $n = 3$

- **Non-leaf**
  - Full node: 120, 150, 180

- **Leaf**
  - Value: 3, 5, 11

- **min. node**
  - Value: 30, 35
B+ Tree Rules (tree of order n)

1. All leaves are at same lowest level (balanced tree)

2. Pointers in leaves point to records, except for “sequence pointer”
B+ Tree Rules (tree of order n)

(3) Number of pointers/keys for B+ tree:

<table>
<thead>
<tr>
<th></th>
<th>Max ptrs</th>
<th>Max keys</th>
<th>Min ptrs→data</th>
<th>Min keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-leaf (non-root)</td>
<td>n+1</td>
<td>n</td>
<td>⌊(n+1)/2⌋</td>
<td>⌊(n+1)/2⌋-1</td>
</tr>
<tr>
<td>Leaf (non-root)</td>
<td>n+1</td>
<td>n</td>
<td>⌈(n+1)/2⌉</td>
<td>⌈(n+1)/2⌉</td>
</tr>
<tr>
<td>Root</td>
<td>n+1</td>
<td>n</td>
<td>2*</td>
<td>1</td>
</tr>
</tbody>
</table>

* When there is only one record in the B+ tree, min pointers in the root is 1 (the other pointers are null)
Insert Into B+ Tree

(a) simple case
   » space available in leaf

(b) leaf overflow

(c) non-leaf overflow

(d) new root
(a) Insert key = 32

\[ n = 3 \]
(a) Insert key = 32

\[ n=3 \]
(a) Insert key = 7

```
\[\begin{array}{c}
3 \\
5 \\
11
\end{array}\]
```

```
\[\begin{array}{c}
30 \\
31
\end{array}\]
```

```
\[\begin{array}{c}
100
\end{array}\]
```

\(n=3\)
(a) Insert key = 7

\[\begin{array}{c}
3 \\
5 \\
\end{array}\]

\[\begin{array}{c}
3 & 11 \\
\end{array}\]

\[\begin{array}{c}
30 & 31 \\
\end{array}\]

\[\begin{array}{c}
100 \\
\end{array}\]
(a) Insert key = 7

\[ n=3 \]
(c) Insert key = 160
(c) Insert key = 160
(c) Insert key = 160

\[ n=3 \]
(c) Insert key = 160
(d) New root, insert 45

\[ n = 3 \]
(d) New root, insert 45

\[ n = 3 \]
(d) New root, insert 45

n=3
(d) New root, insert 45
Deletion from B+tree

(a) Simple case: no example
(b) Coalesce with neighbor (sibling)
(c) Re-distribute keys
(d) Cases (b) or (c) at non-leaf
(b) Coalesce with sibling
   » Delete 50

\[ n=4 \]
(b) Coalesce with sibling

» Delete 50

\[ n = 4 \]
(c) Redistribute keys
  » Delete 50

\[
\text{n=4}
\]
(c) Redistribute keys
   » Delete 50

\[ n = 4 \]
(d) Non-leaf coalesce
– Delete 37

\[ n=4 \]
(d) Non-leaf coalesce
– Delete 37

\[ n=4 \]
(d) Non-leaf coalesce
   – Delete 37
(d) Non-leaf coalesce
  – Delete 37

new root
B+ Tree Deletion in Practice

Often, coalescing is not implemented
  » Too hard and not worth it! (Most datasets just grow in size over time.)
Interesting Problem:

For B+ tree, how large should n be?

n is number of keys / node
Sample Assumptions:

(1) Time to read node from disk is \((S + Tn)\) msec.
Sample Assumptions:

(1) Time to read node from disk is 
   \((S + Tn)\) msec.

(2) Once block in memory, use binary 
    search to locate key: 
    \((a + b \log_2 n)\) msec.

   For some constants \(a, b\); Assume \(a \ll S\)
Sample Assumptions:

(1) Time to read node from disk is 
$(S + Tn)$ msec.

(2) Once block in memory, use binary search to locate key: 
$(a + b \log_2 n)$ msec. 

For some constants $a$, $b$; Assume $a << S$

(3) Assume B+tree is full, i.e., # nodes to examine is $\log_n N$ where $N = #$ records
Can Get:
\[ f(n) = \text{time to find a record} \]
Find $n_{\text{opt}}$ by setting $f'(n) = 0$

Answer is $n_{\text{opt}} = \text{“a few hundred”}$ in practice
Exercise

\[ f(n) = \log_n N \times (S + T n + a + b \log_2 n) \]

\[ S = 14000 \, \mu s \]
\[ T = 0.2 \, \mu s \]
\[ b = 0.002 \, \mu s \]
\[ a = 0 \, \mu s \]
\[ N = 10,000,000 \]
N = 10 Million Records

<table>
<thead>
<tr>
<th>S</th>
<th>14000</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>0.2</td>
</tr>
<tr>
<td>b</td>
<td>0.002</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>10,000,000</td>
</tr>
</tbody>
</table>

find time

n

times in microseconds
N = 100 Million Records

S = 14000
T = 0.2
b = 0.002
a = 0
N = 100,000,000
Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing
Hash Indexes

key $\rightarrow$ \( h(key) \)

Buckets (block sized)

record / ptr

overflow bucket

Chaining is used to handle bucket overflow
Hash vs Tree Indexes

+ $O(1)$ instead of $O(\log N)$ disk accesses

– Can’t efficiently do range queries
Challenge: Resizing

Hash tables try to keep occupancy in a fixed range (50-80%) and slow down beyond that:
» Too much chaining

How to resize the table when this happens?
» **In memory**: just move everything, amortized cost is pretty low
» **On disk**: moving everything is expensive!
Extendible Hashing

Tree-like design for hash tables that allows cheap resizing while requiring 2 IOs / access
Extendible Hashing: 2 Ideas

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

\[
h(K) \rightarrow 00110101
\]

\( i \) will grow over time; the first \( i \) bits of each key’s hash are used to map it to a bucket
Extendible Hashing: 2 Ideas

(b) Use a directory with pointers to buckets

\[ h(K)[0..i] \] to bucket
Example: 4-bit h(K), 2 keys/bucket

Insert 0010
Example: 4-bit $h(K)$, 2 keys/bucket

$$i = \begin{bmatrix} 1 \\ 0001 \\ 1001 \\ 1100 \end{bmatrix}$$

Insert 1010
Example: 4-bit $h(K)$, 2 keys/bucket

Insert 1010
Example: 4-bit $h(K)$, 2 keys/bucket

Insert 1010

New directory
Example

Insert:
0111
0000

\[ i = 2 \]

\[ \begin{array}{c}
00 \\
01 \\
10 \\
11 \\
\end{array} \]

\[ \begin{array}{c}
1 & 0001 \\
2 & 1001 \\
2 & 1010 \\
2 & 1100 \\
\end{array} \]
Example

i = 2

Insert:
0111
0000
Example

Insert:
0111
0000

Example
Example

i = 2

00
01
10
11

0000 2
0001
0111 2
1001 2
1010
1100 2

Note: still need chaining if values of h(K) repeat and fill a bucket
Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing
Motivation

Example: find records where

```
DEPT = "Toy" AND SALARY > 50k
```
Strategy I:

Use one index, say Dept.

Get all Dept = “Toy” records and check their salary
Strategy II:

Use 2 indexes; manipulate pointers

Toy → | | | | | | | | | | Sal ← | | | | | | | | | |
> 50k
Strategy III:

Multi-key index

One idea:
Example

Dept

Index

Art
Sales
Toy

Index

Salary

Index

10k
15k
17k
21k

Example Record

Name=Joe
DEPT=Sales
SALARY=15k
k-d Tree

Splits dimensions in any order to hold k-dimensional data
k-d Tree

```plaintext
CS 245
```
k-d Tree
k-d Tree
k-d Tree

Efficient range queries in both dimensions
Summary

Wide range of indexes for different data types and queries (e.g. range vs exact)

Key concerns: query time, cost to update, and size of index

Next: given all these storage data structures, how do we run our queries?