Data Storage & Indexes

Instructor: Matei Zaharia
Outline

C-Store (co-designing compute & storage)

Indexes
Outline

C-Store (co-designing compute & storage)

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>asArray()</td>
<td></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

\[
\begin{align*}
\text{COUNT}(\text{Column } c1) \\
\quad b &= \text{get next compressed block from } c1 \\
\quad \text{while } b \text{ is not null} \\
\quad \quad \text{if } b.\text{isOneValue()} \\
\quad \quad \quad x &= \text{fetch current count for } b.\text{getStartVal()} \\
\quad \quad \quad x &= x + b.\text{getSize()} \\
\quad \quad \text{else} \\
\quad \quad \quad a &= b.\text{asArray()} \\
\quad \quad \quad \text{for each element } i \text{ in } a \\
\quad \quad \quad \quad x &= \text{fetch current count for } i \\
\quad \quad \quad x &= x + 1 \\
\quad b &= \text{get next compressed block from } c1
\end{align*}
\]

Figure 2: Pseudocode for Simple Count Aggregation
Figure 4: Compressed column sizes for varied compression schemes on column with sorted runs of size 50 (a) and 1000 (b)
Performance: Compressed Eval

(a) Sorted runs of length 50
(b) Sorted runs of length 1000

How would the results change on SSDs?
Figure 10: Decision tree summarizing our results regarding the proper selection of compression scheme.
Outline

C-Store (co-designing compute & storage)

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

<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>
<tr>
<td>70</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>
### 2-level sparse index

<table>
<thead>
<tr>
<th>10</th>
<th></th>
<th></th>
<th>10</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td></td>
<td></td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td></td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td></td>
<td></td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>330</td>
<td></td>
<td></td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>410</td>
<td></td>
<td></td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>490</td>
<td></td>
<td></td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>570</td>
<td></td>
<td></td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>

### Sequential File

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

---

File and 2\textsuperscript{nd} level index blocks need not be contiguous on disk
Sparse vs Dense Tradeoff

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

**Dense:** Can tell whether a key is present 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 1\textsuperscript{st} 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

\[\begin{array}{c}
10 \\
30 \\
50 \\
70 \\
90 \\
110 \\
130 \\
150 \\
\end{array}\]
Deletion: Sparse Index

– delete record 40
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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
</tr>
<tr>
<td>130</td>
</tr>
<tr>
<td>150</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
</tr>
</tbody>
</table>
Deletion: Sparse Index

- delete record 30

| 10 | 10 |
| 30 | 20 |
| 50 | 30 |
| 70 | 40 |
| 90 | 50 |
| 110| 60 |
| 130| 70 |
| 150| 80 |
Deletion: Sparse Index
– delete record 30
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 Diagram**

<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>50</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>
Deletion: Dense Index

– delete record 30

```
| 10 | 10 |
| 20 | 20 |
| 30 | 30 |
| 40 | 40 |
| 50 | 50 |
| 60 | 60 |
| 70 | 70 |
| 80 | 80 |
```
Deletion: Dense Index

- delete record 30

```
10  |  10
20  |  20
30  |  30
40  |  40
50  |  50
60  |  60
70  |  70
80  |  80
```
Deletion: Dense Index

- delete record 30
Insertion: Sparse Index

– insert record 34
Insertion: Sparse Index
– insert record 34

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

– insert record 15
Insertion: Sparse Index

- insert record 15

```
10
30
40
60
```

```
10
20 15
30
30 20
40
50
60
```
Insertion: Sparse Index

- insert record 15

• Illustrated: Immediate reorganization

• Variation:
  - insert new block (chained file)
  - update index
Insertion: Sparse Index

– insert record 25

<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>
<tr>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>60</td>
</tr>
</tbody>
</table>
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>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ordering field:

<table>
<thead>
<tr>
<th></th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>20</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>70</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>80</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>40</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>100</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>10</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>90</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>60</th>
</tr>
</thead>
</table>
Secondary Indexes

Sparse index:

does not make sense!
Secondary Indexes

Dense index:

Ordering field:

- 30
- 50
- 20
- 70
- 80
- 40
- 100
- 10
- 90
- 60
Secondary Indexes

Dense index:

Sparse higher level

Ordering field
Duplicate Values in Secondary Indexes

10
20
30
40
50
60...

buckets

20
10
40
20

10
40

10
40

30
40
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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>DeptID</th>
<th>FloorID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<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 (2nd floor)

Intersect “Toy” bucket and “2nd floor” buckets to get list of matching employees
Often Used in Text Search

Documents

...the cat is fat ...

...was raining cats and dogs...

...Fido the dog ...
Often Used in Text Search

Documents

...the cat is fat ...

...was raining cats and dogs...

...Fido the dog ...

Inverted lists
Extension: 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
150 156 179
180 200
Sample Non-Leaf

- 57
- 81
- 95

- to keys
- to keys
- to keys
- to keys

- < 57
- 57 ≤ k < 81
- 81 ≤ k < 95
- ≥ 95
Sample Leaf Node

From non-leaf node

to next leaf in sequence

To record with key 57

To record with key 81

To record with key 95
Size of Nodes on Disk

\[
\begin{align*}
&\{ & n + 1 \text{ pointers} \\
&\{ & n \text{ keys} \\
(\text{Fixed size nodes})
\end{align*}
\]
Don’t Want Nodes to be Too Empty

Use at least

Non-leaf: \[\lceil (n+1)/2 \rceil \] pointers

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

Non-leaf

Leaf

Full node

min. node
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><strong>Non-leaf (non-root)</strong></td>
<td>n+1</td>
<td>n</td>
<td>⌈(n+1)/2⌉</td>
<td>⌈(n+1)/2⌉-1</td>
</tr>
<tr>
<td><strong>Leaf (non-root)</strong></td>
<td>n+1</td>
<td>n</td>
<td>⌊(n+1)/2⌋</td>
<td>⌊(n+1)/2⌋</td>
</tr>
<tr>
<td><strong>Root</strong></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

n=3
(a) Insert key = 7

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

n=3

CS 245
(c) Insert key = 160

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

n=3

100

120 150 180

150 156 179

160 179

180 200
(c) Insert key = 160

n = 3
(c) Insert key = 160

\[ \begin{array}{c}
100 \\
120 \\
150 \\
160 \\
156 \\
150 \\
180
\end{array} \]
(d) New root, insert 45

```
n=3
```

```
1 2 3
10 12
20 25
30 32 40
```
(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
(c) Redistribute keys

» Delete 50

n=4
(c) Redistribute keys

» Delete 50

n=4
(d) Non-leaf coalesce
– Delete 37
(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 only tend to grow in size over time.
Interesting Problem:

For B+ tree, how large should n be?

\[ n \text{ is number of keys / node} \]

With modern hardware, get \( n = 1000 \) or more
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

\[ h(K)[0..i] \rightarrow \vdots \rightarrow \text{to bucket} \]
Example: 4-bit $h(K)$, 2 keys/bucket

$$i = \begin{array}{c}
1 \\
\text{local depth}
\end{array}$$

$$\text{global depth}$$

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

\[ i = \begin{array}{c}
1 \\
0001
\end{array} \]

Insert 1010

Diagram:
- Global depth
- Local depth
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
Example

Insert:

0111
0000
Example

Insert:

0111
0000
Example

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

$\text{DEPT} = \text{"Toy" \ AND \ SALARY > 50k}$
Strategy I:

Use one index, say Dept.

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

\[ I_1 \]
Strategy II:

Use 2 indexes; manipulate pointers

Toy →  ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| ||| |
Strategy III:

Multi-key index

One idea:
Example

<table>
<thead>
<tr>
<th>Dept</th>
<th>Name=Joe</th>
<th>DEPT=Sales</th>
<th>SALARY=15k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sales</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Toy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example Record

<table>
<thead>
<tr>
<th>Salary Index</th>
<th>10k</th>
<th>15k</th>
<th>17k</th>
<th>21k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12k</td>
<td>15k</td>
<td>15k</td>
<td>19k</td>
</tr>
</tbody>
</table>

Dept Index

Art
Sales
Toy
k-d Tree

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

10  20

a
b
c
d
e
f
h
i
j
k
l
m
n
10  20
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?
Storage System Examples

**MySQL**: transactional DBMS

» Row-oriented storage with 16 KB pages
» Variable length records with headers, overflow
» Index types:
  • B-tree
  • Hash (in memory only)
  • R-tree (spatial data)
  • Inverted lists for full text search
» Can compress pages with Lempel-Ziv
Apache Parquet + Hive: analytical data lake

» Column-oriented storage as set of ~1 GB files (each file has a slice of all columns)

» Various compression and encoding schemes at the level of pages in a file
  • Special scheme for nested fields (Dremel)

» Header with statistics at the start of each file
  • Min/max of columns, nulls, Bloom filter

» Files partitioned into directories by one key