Physical Storage and Indexes

Types

Heap - Tuples are stored in a random order. Best when retrieving the whole file.

Sorted File - Best when records must be retrieved in order (ranges)

Indexes - Can be trees or hashes. Quick update, find subset based on search key.

Index Storage

Alternative 1 - `<key, whole record>`

Alternative 2 - `<key, id of matching record>, <key, id>, ...`

Alternative 3 - `<key, rid1, rid2, rid3, ...>`

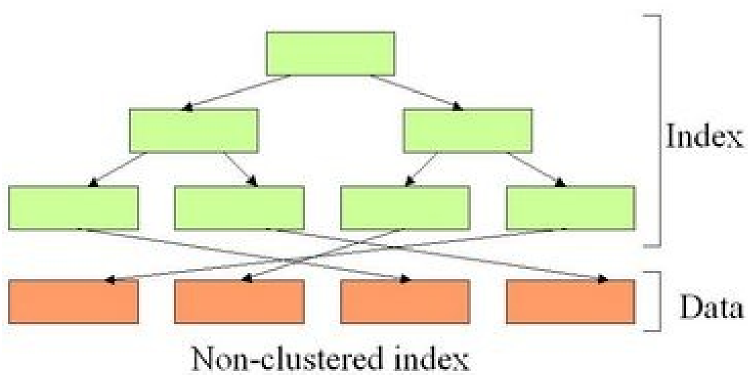
Alternative 1

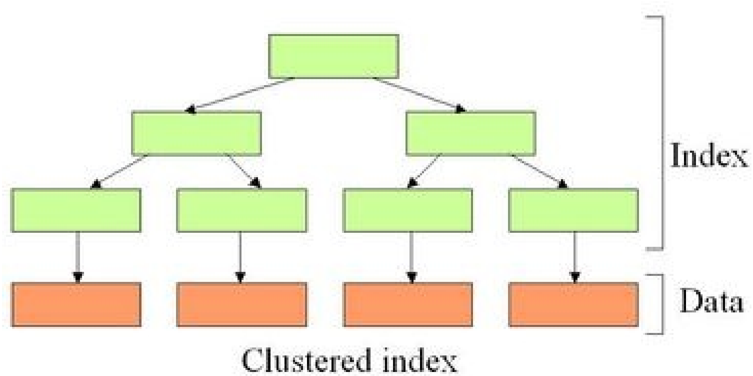
In this case, the index is used to organize the records on disk instead of a randomly ordered heap file. At most one index can use alternative 1 since records can't be sorted in more than one way.

Index Classification

Primary vs. Secondary - An index is primary if it contains a primary key which is unique.

Clustered vs. Unclustered - An index is clustered if the order of the entries is the same or close to the order of the records on disk. Clustered implies Alternative 1 and Alternative 1 implies clustered. Cost varies greatly for looking up a record based on whether the index is clustered or unclustered.





Cost Model Analysis

Background

(a) Heap file

(b) Sorted file

(c) Clustered B+ Tree in Alternative 1 - Since the pages are only 67% full, there will be 1.5P data pages in this index. The height of this tree would be logF (1.5P).

(d) Unclustered B+ Tree - Since this is stored in either alternative 2 or 3, the size of this is only 10% of what it would normally be. Since B+ trees have 67% occupancy, the total data pages would be 0.15P. But, since the records are stored in alternative 2 or 3, we need to store the real data somewhere else. So there must be an additional P data pages elsewhere. The height would be LogF (0.15P)

(e) Unclustered Hash Table - P is in a heap file. 0.125P data pages in the index buckets since Hash tables have roughly 80% occupancy per page.

Variables

```
 P = # of data pages
R = # of records / page
D = Avg. time to read or write a page
M = # of matching pages
m = # of matching records
```

Comparison

```
    | Full Scan     | = (Unique)        | = (Non-unique)     | Range             |
(a) | P(D)          | 0.5PD             | PD                 | PD                |
(b) | P(D)          | log2(P)D          | log2(P)D + M       | log2(P)D + MD     |
(c) | 1.5P(D)       | logF(1.5P)D       | logF(1.5P)D + MD   | logF(1.5)D + MD   |
(d) | (0.15P+PR)D   | logF(0.15P)D + D  | logF(0.15P)D + mD  | logF(0.15)D + MD  |
(e) | (0.125P+PR)D  | D + D             | D + mD             | D + mD            |
```

Indexes

B+ Tree Index

Supports both equity and range searches. Leaf nodes are chained together like a linked list.

Insert Delete

Costs `logF(N)` where `F` is the fanout of the nodes and `N` is the number of leaf pages.

Fanout is the number of pointers out of the node.

Each node has a minimum 50% occupancy except the root node. This means that each node has `d <= m <= 2d` entries where `d` is the order of the tree.

```
Average Entries =  (2d * occupancy - 1)^height
```

Example: order 100 tree with occupancy 67% and height 3

```
(2*100 * .67)^3 = (134 - 1)^3 = 2,352,637
```

Hash Index

Best for equality searches. Doesn't support range search.

We can use a directory structure to hash values based on their least significant digits.

Definitions

```
Directory Size - length of array containing pointers to buckets
Global Depth - # of least significant bits mapping to buckets
Local Depth - # of least significant bits mapping to this bucket
Bucket Size - max length of a bucket array
```

Insert

If a bucket is full, add one to the local depth, split it and redistribute the nodes. Doubling the directory size may be necessary.

Evaluating Relational Operators

Background

Schema:

```
Sailors (S) {
  sid: integer
  sname: string
  rating: integer
  age: real
}

Reserves (R) {
  sid: integer
  bid: integer
  day: date
  rname: string
}
```

Size:

```
Sailors:
  50 bytes / tuple
  80 tuples / page
  500 pages
```

```
Reserves:
  40 bytes / tuple
  100 tuples / page
  1000 pages
```

Selection

Output size: `# of Tuples * Reduction Factor`

If we don't use an index, we typically need to scan all the records in a relation. (Cost = `# of pages in R`)

Two main approaches:

1. **Most Selective Path** - Pick the selection that reduces the number of tuples by the largest factor and then scan through the results discarding tuples that don't match other selectors.
2. **Intersection of RIDs** - Use multiple indexes to pick out the RIDs of all matching records for each selection. The take the union of the RIDs to get the final set.

Joins

Consider this query:

```
 SELECT *
FROM S, R
WHERE R.sid = S.sid
```

Variables:

```
 M = # of pages in R
N = # of pages in S
Pr = # of tuples / page in R
Ps = # of tuples / page in S
```

Simple Nested Loops Join

Algorithm:

```
 for x in R1
  for y in R2
    if x.sid = y.sid then add <x, y>
```

Cost:

```
 M + Pr * M * N
= 1000 + 100 * 1000 * 500
= 50,001,000
```

Page Nested Loop

Algorithm:

```
for page_x in R1
   for page_y in R2
     write <x,y> where X is in page_x and Y is in page_y
```

Cost:

```
M + M * N
= 1000 + 1000 * 500
= 501,000
```

Index Nested Loop

If an index exists on the join column of one relation, we can make it the inner relation and exploit the index.

Cost:

```
M + (M * Pr * Cost of Index)
```

Cost of Index:

```
Probing:
~ 1.2 I/Os for Hash Table
~ 2 - 3 I/Os for B+ Tree

Clustered: 1 I/O + probing cost
Unclusterd: # of records + probing cost
```

Ex: Unclustered Hash (Alt 2)

```
1000 pages * 100 tuples / page * (1.2 + 1)
```

Block Nested Loop

Use one buffer page as input buffer, one pages as output buffer, and the rest to hold the outer 'block'.

Cost:

```
Scan of Outer + # of outer blocks * Scan of Inner

# of outer blocks = Ceil(# of pages / block size)
```

Ex. 100 page blocks

```
M + (M / 100) * N
1000 + 1000 / 100 * 500
```

Sort Merge Join

Sort both R & S on the join column, then progressively scan R and S for matches.

Cost:

```
M * log(M) + N * log(N) + (M + N)
```

Hash Join

Partition both relations using hash function H. R tuples in partition i will only match S tuples in partition i.

Read in a partition of R and hash it using H2. Scan the matching partition in S and search for matches.

```
k = # of partitions
  = B - 1
```

Cost:

```
 Partitioning: 2(M + N)
Reading:        (M + N)
Total:        = 3(M + N)
```

External Merge Sort

To sort a file with N pages using B buffer pages, we can use a multi-way merge sort.

Algorithm:

```
 Pass 0: use B buffer pages. Produce `Ceil(N/B)` sorted runs of B pages
Pass 1..X: merge B - 1 runs
```

Calculations:

```
 # of Runs    = Ceil(N/B)
# of Passes = 1 + Ceil(logB-1(# of Runs))

Cost = 2 * N * # of Passes
```

Ex: B = 5, N = 108

```
 # of Passes = 1 + Ceil(log4(108 / 5))
            = 1 + Ceil(log4(22))
            = 1 + 3
            = 4

Pass 0: [Ceil(108/5)] 22 sorted runs of 5 pages
Pass 1: [Ceil(22/4)] 6 sorted runs of 20 pages
Pass 2: [Ceil(6/4)] 2 sorted runs of 80 pages
Pass 3: Sorted file of 108 pages
```

Query Optimization

System R

Only considers left-deep plans because they can be pipelined together. Avoids Cartesian products because they are completely inefficient.

Reduction Factors

```
Resulting Cardinality = Max Tuples * Product of all RFs
```

Assuming all terms are independent, we can use these identities:

- **col = value**: 1 / # of Keys
- **col1 = col2**: 1 / Max(# of Keys in col 1 or 2)
- **col > value**: (High Value - value) / (High Value - Low Value)

Cost of Single Relation Plans

Primary Key Selection:

```
 Tree: Height + 1
Hash: 1.2
```

Clustered Index:

```
(# of Index Pages + # of Relation Pages) * RFs
```

Unclustered Index:

```
(# of Index Pages + # of Tuples) * RFs
```

Sequential Scan:

```
# of Relation Pages
```

Multidimensional Indexes

Hashes and trees are really 1 dimensional so we can use z curves and hilbert curves to map spacial data (n dimensional).

Grid File

Dynamic version of multi-attribute hashing that adapts to non-uniform distributions. Each cell links to one disk page which means 2 I/Os per exact match query.