Data Storage and Retrieval – Assignment 8

Aaron Niskin

October 28, 2016

1. **Almasri & Navathe, Exercise 17.18, parts a, b, d, h**

   Consider a disk with block size B = 512 bytes. A block pointer is P = 6 bytes long, and a record pointer is P_R = 7 bytes long. A file has r = 30,000 EMPLOYEE records of fixed length. Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Department_code (9 bytes), Address (40 bytes), Phone (10 bytes), Birth_date (8 bytes), Sex (1 byte), Job_code (4 bytes), and Salary (4 bytes, real number). An additional byte is used as a deletion marker.

   a. **Calculate the record size R in bytes.**

   ```
   record_size = 30 + 9 + 9 + 40 + 10 + 8 + 1 + 4 + 4 + 1
   record_size
   ```

   ```
   ## [1] 116
   ```

   b. **Calculate the blocking factor bfr and the number of file blocks b, assuming an unspanned organization.**

   ```
   block_size = 512
   bfr = floor(block_size / record_size)
   bfr
   ```

   ```
   ## [1] 4
   ```

   As for the number of blocks,

   ```
   num_records = 30000
   unspanned_num_blocks = ceiling(num_records / bfr)
   unspanned_num_blocks
   ```

   ```
   ## [1] 7500
   ```

   c. Needed for part D

   Suppose that the file is ordered by the key field Ssn and we want to construct a primary index on Ssn. Calculate

   (i) **The index blocking factor bfr_i (which is also the index fan-out fo);**

   ```
   ssn_index_record_size = 6 + 9 # block pointer + ssn
   ssn_index_record_size
   ```

   ```
   ## [1] 15
   ```

   ```
   ssn_index_bfr = floor(block_size / ssn_index_record_size)
   ssn_index_bfr
   ```

   ```
   ## [1] 34
   ```

   (ii) **The number of first-level index entries and the number of first-level index blocks;** Since our records are sorted by `ssn`, we need only one record per block in our index. Hence the number of required index records is the number of record blocks, 7500.

```
ssn_index_num_blocks = ceiling(unspanned_num_blocks / ssn_index_bfr)
ssn_index_num_blocks
```

```
## [1] 221
```

(iii) **The number of levels needed if we make it into a multilevel index;**
Technically, we should do an iterative thing where we take the ceiling of the log every time and wait till we get 1, but that's exactly what I did! (Did I get you there for a second?) But as long as the distance from the number of records to a power of the bfr is greater than the log below, the log would give an accurate measure. If not, we'd be off by 1. But again, this won't be an issue because I changed the code (to account for the next part).

```
get_levels <- function(bfr, num_blocks){
  ### We create a list and an accumulator
  tmp <- c(ceiling(num_blocks / bfr))
  i = 2
  while (tmp[i-1] != 1) {
    tmp[i] <- ceiling(tmp[i-1]/ bfr)
    i = i + 1
  }
  return(tmp)
}
ssn_index_levels = get_levels(ssn_index_bfr, unspanned_num_blocks)
sum(ssn_index_levels > 0)
```

```
## [1] 3
```

(iv) **The total number of blocks required by the multilevel index;**

```
sum(ssn_index_levels)
```

```
## [1] 229
```

(v) **The number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the primary index.**
Assuming they're not using a b-tree with some pointers in each level, we will have to access 4 blocks. 3 for the index levels and 1 for the record itself.

d. **Suppose that the file is not ordered by the key field Ssn and we want to construct a secondary index on Ssn. Repeat the previous exercise (part c) for the secondary index and compare with the primary index.**

(i) **The index blocking factor bfr_i (which is also the index fan-out fo);**
These values won't change due to the lack of order in this column:

```
ssn_index_record_size
```

```
## [1] 15
```

```
ssn_index_bfr
```

```
## [1] 34
```

(ii) **The number of first-level index entries and the number of first-level index blocks;**
Since our records are not sorted by `ssn`, we need one record in our index for every record in the data. Hence the number of required index records is the number of data records, which R-Studio does not want me to show you, for some reason. Secrecy is an issue, I suppose.

```
ssn_index_num_blocks = ceiling(num_records / ssn_index_bfr)
ssn_index_num_blocks
```

```
## [1] 883
```

(iii) **The number of levels needed if we make it into a multilevel index;**
We can use the functions we defined earlier! Yay programming!

```
ssn_index_levels = get_levels(ssn_index_bfr, num_records)
sum(ssn_index_levels > 0)
```

```
## [1] 3
```

(iv) **The total number of blocks required by the multilevel index;**

```
sum(ssn_index_levels)
```

```
## [1] 910
```

(v) **The number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the primary index.**
Assuming they're not using a b-tree with some pointers in each level, we will have to access 4 blocks. 3 for the index levels and 1 for the record itself.

e. Not required for the problem

f. Not required for the problem

g. **Suppose that the file is not ordered by the key field Ssn and we want to construct a B^+-tree access structure (index) on Ssn. Calculate**

(i) **The orders p and p_{leaf} of the B^+-tree**

$$\begin{aligned} 6p + 9(p-1) &\leq 512 \\ 15p - 9 &\leq 512 \\ 15p &\leq 521 \\ p &\leq 34.7333333 \\ p &= 34 \end{aligned} \qquad \begin{aligned} 6 + p_{leaf}(9+7) &\leq 512 \\ 6 + p_{leaf}(16) &\leq 512 \\ p_{leaf}(16) &\leq 506 \\ p_{leaf} &\leq 31.625 \\ p_{leaf} &= 31 \end{aligned}$$

(ii) **The number of leaf-level blocks needed if blocks are approximately 69% full (rounded up for convenience)**
That should just be the number of records divided by the number of records in each block:

```
actual_p = ceiling(.69 * 34)
actual_p
```

```
## [1] 24
```

```
actual_p_leaf = ceiling(.69 * 31)
actual_p_leaf
```

```
## [1] 22
```

```
num_leaves = ceiling(num_records / actual_p_leaf)
num_leaves
```

```
## [1] 1364
```

(iii) **The number of levels needed if internal nodes are also 69% full (rounded up for convenience)**

```
levels = get_levels(actual_p, num_leaves)
levels
```

```
## [1] 57  3  1
```

```
sum(levels > 0)
```

```
## [1] 3
```

(iv) **The total number of blocks required by the B^+-tree**

```
sum(levels) + num_leaves
```

```
## [1] 1425
```

(v) **The number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the B^+-tree.**
We would need 5. 3 for the B^+-tree, 1 for the leaf, and 1 for the record itself.

h. **Repeat part g, but for a B-tree rather than for a B^+-tree. Compare your results for the B-tree and for the B^+-tree.**

(i) **The orders p and p_{leaf} of the B^+-tree**

$$\begin{aligned} 6p + (9+7)(p-1) \leq & 512 \\ 22p - 16 \leq & 512 \\ 22p \leq & 528 \\ p \leq & 24 \\ p = & 24 \end{aligned}$$

(ii) **The number of leaf-level blocks needed if blocks are approximately 69% full (rounded up for convenience)**

```
get_new_levels <- function(bfr, num_blocks){
  ### We create a list and an accumulator
  tmp <- c(1)
  num_recs = num_blocks - bfr + 1
  i = 2
  while (num_recs > 0) {
    ### Each time we point to bfr - 1 many records
    this_round <- tmp[i-1] * bfr
    if (this_round * bfr >= num_recs) {
      tmp[i] = ceiling(num_recs / bfr)
      num_recs = 0
    }
    else {
      # Here we're probably going to have a less than half full
      # leaf node layer. Meaning that the i-1 layer is probably
      # going to be less than half full. This violates our assurances,
      # but this is only an estimation.
      tmp[i] = this_round
      num_recs = num_recs - this_round *(bfr - 1)
    }
    i = i + 1
  }
  return(tmp)
}
actual_p = ceiling(.69 * 24)
actual_p
```

```
## [1] 17
```

```
levels = get_new_levels(actual_p, num_records)
```

This approximation of the number of leaf nodes is seriously underestimated, but it seems to me as though the number of leaf nodes is not very likely to be an issue of much importance.

```
levels[length(levels)]
```

```
## [1] 1476
```

(iii) **The number of levels needed if internal nodes are also 69% full (rounded up for convenience)**

```
sum(levels > 0)
```

```
## [1] 4
```

(iv) **The total number of blocks required by the B^+-tree**
Since there is no real distinction (in the size, anyway) of leaves as opposed to internal nodes:

```
sum(levels)
```

```
## [1] 1783
```

(v) **The number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the B^+-tree.**
We would need 6. 4 for the B^+-tree, 1 for the leaf, and 1 for the record itself.

2. **Relational algebra and query plans.**

a. **Write the MillionSongs database query**

```
SELECT artist_name, title, year
  FROM tracks t, artists a
  WHERE t.artist_id = a.artist_id
    AND year < 2000;
```

in relational algebra.

$$\Pi_{artist\_name,title,year}\left(\sigma_{\text{year} < 2000}\left(\text{tracks} \bowtie_{\text{tracks.artist\_id} = \text{artists.artist\_id}} \text{artists}\right)\right)$$

a. **Show two query plans, including the one Postgres chooses.**
The one Postgres chooses:

```
Hash Join  (cost=120.48..442.02 rows=2076 width=38)
  Hash Cond: (t.artist_id = a.artist_id)
  ->  Seq Scan on tracks t  (cost=0.00..293.00 rows=2076 width=43)
      Filter: (year < 2000)
  ->  Hash  (cost=71.88..71.88 rows=3888 width=33)
      ->  Seq Scan on artists a  (cost=0.00..71.88 rows=3888 width=33)
```

Another, less awesome one:

```
Seq Scan
->  Filter (t.year < 2000)
    ->  Join
        ->  Cond (t.artist_id = a.artist_id)
            ->  Index Scan on artists a
            ->  Seq Scan on tracks t
```

b. **Calculate the costs of two query plans using Postgres's default cost config parameters.**

```
SELECT relpages, reltuples
  FROM pg_class
  WHERE relname = 'tracks';

 relpages | reltuples
----------+-----------
      168 |     10000
(1 row)
```

Cost for scan on tracks

```
tmp1 <- 1.0 * (168) + 0.01 * (10000)
```

```
SELECT relpages, reltuples
  FROM pg_class
  WHERE relname = 'artists_pkey';

 relpages | reltuples
----------+-----------
       22 |      3888
(1 row)
```

Cost index scan on artists

```
tmp2 <- 1.0 * (22) + 0.01 * (3888)
```

Filter cost

```
SELECT avg_width FROM pg_stats WHERE tablename = 'artists';
 avg_width
-----------
        19
        14
         4
         4
(4 rows)
SELECT avg_width FROM pg_stats WHERE tablename = 'tracks';
 avg_width
-----------
        19
        20
        19
        20
        19
         4
         4
(7 rows)
```

```
tmp = 19 + 14 + 4 + 4 + 19 + 20 + 19 + 20 + 19 + 4 + 4
num_pages_join = ceiling(10000 * tmp / 8000)
```

So the approximate number of pages is 183. Hence, the approximate cost of selecting those with year > 2000

```
tmp3 <- 1.0 * (num_pages_join) + 0.01 * (10000) + 0.0025 * (10000)
```

Selecting just the columns we want: FREE!

So something around 636.88?

c. **For the query plan Postgres chose, compare your cost to what explain says.**

It's about 1.5 times as big as their estimation, so that seems somewhat reasonable to me considering what we're doing and the hash costs incurred by their version.