CS 377: Database Systems

Homework #4 Solutions

1. **Extensible Hashing** (5 + 5 × 3 points)

   Assume we have the following records where we indicate the hashed key (binary representation) in brackets. We want to use an Extendible Hashing structure where each bucket can hold up to 3 records and the structure initially starts out empty (one empty bucket).

   - 'a' [00001]
   - 'b' [01000]
   - 'c' [00000]
   - 'd' [10100]
   - 'e' [10101]
   - 'f' [00010]
   - 'g' [11000]
   - 'h' [10001]

   Consider the result after the records above have been inserted in the order shown, using the highest-bits for the hash function. In other words, records in a bucket of local depth d, agree on their *leftmost* d bits.

   (a) What does the hash structure look like after inserting all the records?
   (b) What is the global depth of the resulting directory?
   (c) What record causes the first split?
   (d) What record causes the second split?
   (e) After inserting these records, suppose that we insert 'i' [10000]. How many buckets will we have now?
   (f) What other keys are in the same bucket with record 'i'?

   (**ANSWER**)

   (a) i. The initial state is a single empty bucket with 3 slots, so we fill it with items a, b, and c.
   ii. When we go to insert record 'd' into the picture, we find the bucket is full, so we need to increase the number of bits that we use from the hash value. We now use 1 bit, allowing 2 buckets. Since we split the bucket, we place in the new bucket those records whose search key has a hash value beginning with 1 and leaving the original bucket the other records. In other words, a, b, and c now are in 1 bucket, and d is in a separate bucket.
   iii. We add e to the same bucket as d since there is still space.
   iv. When we go to add f, we find that the bucket with the hash value beginning with 0 is full, so we need to increase the hash bits to 2 bits. Since we are not splitting on 1, the two entires of the bucket address 10 and 11 both point to the hash bucket that was originally 1. For the other bucket that was split, we separate the items appropriately, such that a, c and f go into one bucket (00) and b goes into the other bucket (01).

v. Adding g is straightforward as there is still space in the third bucket (10 and 11).

vi. Adding h causes another split as the third bucket is now full, so we create the 4th bucket and update the bucket addresses to point to the right bucket.

Figure 1 shows the resulting hash structure after inserting these records.

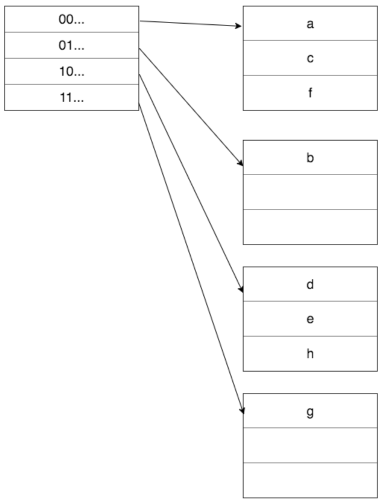

Figure 1: Hash structure after inserting records a-h

(b) The global depth of the resulting directory is 2 (number of bits to represent the prefix).

(c) The record that causes the first split is (d).

(d) The record that causes the second split is (f).

(e) Based on the hash structure shown in Figure 1, we can see that when we try to insert i, there is a not enough space in the (10) bucket, so that means we will introduce a bucket and increase our hash bit prefix to 3 bits (global depth of 3). Thus we have 5 buckets in our hash directory.

(f) Since we have split bucket (10) into (101) and (100), i is in the same bucket as h.

2. **B$^+$-Tree** (20 points)

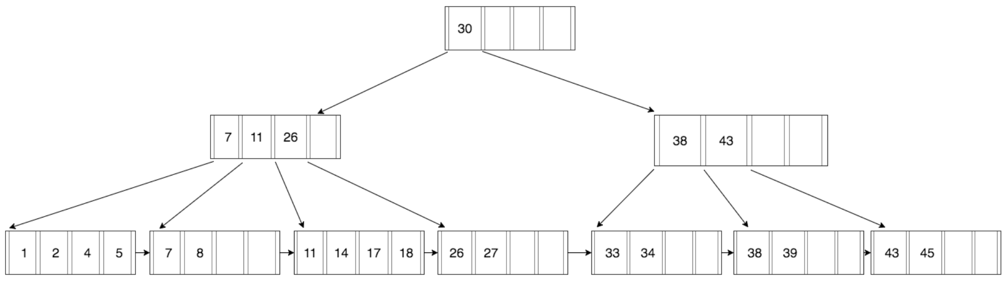

Figure 2: B$^+$-Tree for problem 2

Consider the B$^+$-Tree of order d = 2, height h = 3 levels, and number of values per node n =4 shown in Figure 2. For each subpart, draw the tree after the specified operation starting from the one (Figure 2) provided below (i.e., the subparts are independent and do not affect one another). Please follow the guidelines listed below:

- The left pointer denotes $<$ while the right pointer denotes $\geq$.

- In the case of underflow, if you can borrow from both siblings, choose the one on the *left*.
- In the case of overflow, promote the key in the middle from the left sibling.

(a) Insert tuple with search key value 10.
(b) Insert tuple with search key value 20.
(c) Delete tuple with search key value 7.
(d) Delete tuples with search key values 17, 18, 26.

(**ANSWER**)

(a) The insert is fairly straightforward as the node with 7, 8 still have space for 10. The new node is colored in orange.

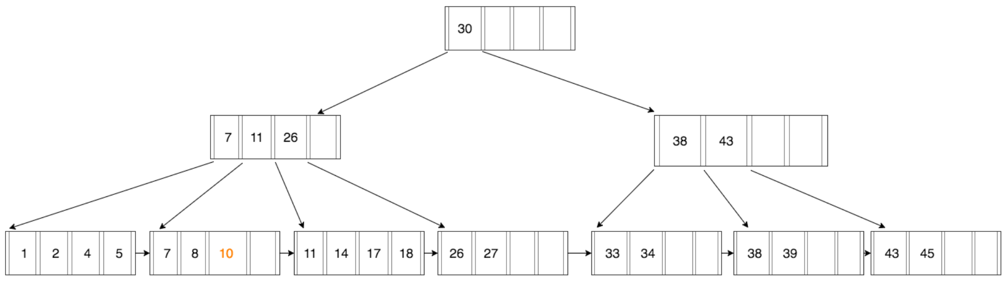

Figure 3: B$^+$-Tree for problem 2(a)

(b) We want to insert it into leaf node 3 (starting with 11, 14, 17, 18), but unfortunately this one is full so we need to split it into 2 nodes. So we first split the node into (11, 14) and (17, 18), then add 20 to the end of (17, 18). Since we introduced the new node, we need to update the parent node to propagate pointer to the leaf node, and since there is space we can easily insert it. See Figure 4 for the new tree with the changes colored in orange.

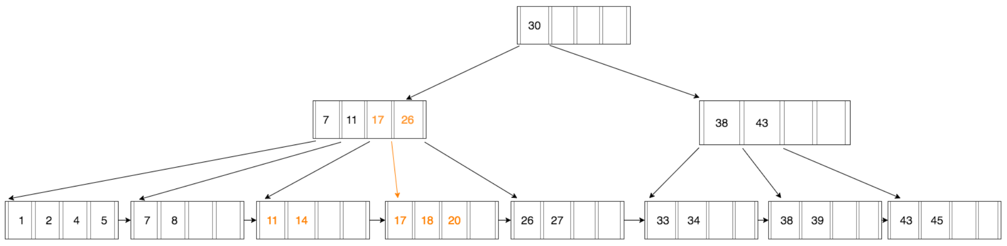

Figure 4: B$^+$-Tree for problem 2(b)

(c) Since we are deleting 7 from the second node from the left (leaf with 7,8), that leaves 8 as the only node which means that it violates the minimum occupancy. Thus we need to promote items from the left sibling. Based on the guidelines stated above, we will promote 4 as the new start of the node, resulting in Figure 5.

(d) Since we are deleting 7 from the second node from the left (leaf with 7,8), that leaves 8 as the only node which means that it violates the minimum occupancy. Thus we need to promote items from the left sibling. Based on the guidelines stated above, we will promote 4 as the new start of the node, resulting in Figure 6.

3. **Using Indexes on Yelp Reviews** (10 points) Consider the following Yelp review database:

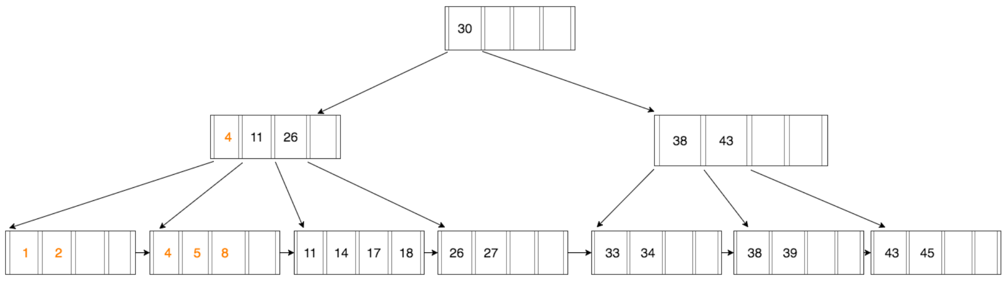

Figure 5: B$^+$-Tree for problem 2(c)

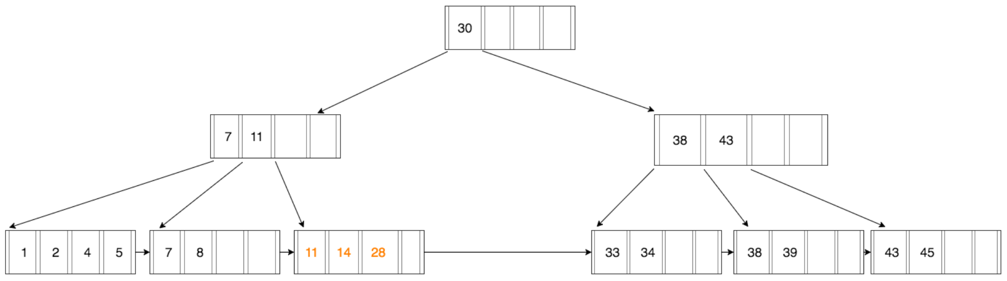

Figure 6: B$^+$-Tree for problem 2(d)

```
Business(bid, name, city, state)
BusinessCategory(bid, category)
User(uid, name, fans)
Review(bid, uid, stars, date, votes_useful, votes_funny, votes_cool)
```

In addition to the primary keys that have been specified, we've also created an index on the attribute city on the relation business using the following command:

```
CREATE INDEX bcity_idx ON business(city);
```

For the following queries, answer if the index we created was used or not.

(a) `SELECT * FROM business`
    `WHERE city = 'Atlanta';`

(b) `SELECT * FROM business`
    `WHERE city > 'Atlanta';`

(c) `SELECT * FROM business`
    `WHERE city > 'B' AND city < 'G' AND name = 'A';`

(d) `SELECT * FROM business`
    `WHERE city > 'B' AND city < 'G' OR state = 'GA';`

(e) `SELECT * FROM business`
    `WHERE city != 'Atlanta';`

(**ANSWER**)

(a) Yes the index will be used.

(b) Yes the index will be used since it's a B$^+$-Tree and we can do range search.

(c) Yes the index will be used since it's a B$^+$-Tree and we can do range search to find the city before pruning the tuples for the name match.

(d) No since the there is an OR condition and that does not have an index.

(e) No, the index will not be used since it's not equal.

4. **Selection Search via B$^+$-Tree** (15 points)

Consider the following bank database (it was used as an example during the midterm review):

```
Branch(bname, bcity, assets)
Customer(cname, cstreet, ccity)
Loan(lID, bname, amount)
Borrower(cname, lID)
Account(aID, bname, balance)
Depositor(cname, aID)
```

Suppose a B$^+$-Tree index was created on bcity for relation Branch, and there are no other indexes in the database. Describe the optimal selection algorithm, specifying whether the index on bcity was used or not:

(a) $\sigma_{\text{NOT(bcity<‘Brooklyn’)}}(\text{Branch})$

(b) $\sigma_{\text{NOT(bcity=‘Brooklyn’)}}(\text{Branch})$

(c) $\sigma_{\text{NOT(bcity<‘Brooklyn’ OR assets<5000)}}(\text{Branch})$

(**ANSWER**)

(a) Use the index to locate the first tuple whose branch city has the value "Brooklyn". From this tuple, follow the pointer chain until the end to retrieve all the tuples that are greater than or equal to Brooklyn.

(b) The index serves no purpose so we need to scan the file sequentially (linear scan) and select all the tuples whose branch city is not Brooklyn.

(c) Note that this query is equivalent to $\sigma_{(\text{bcity}\geq\text{‘Brooklyn’ AND assets}\geq 5000)}(\text{Branch})$ by De Morgan's laws, NOT (p AND q) $\Leftrightarrow$ NOT(p) AND NOT(q). We can use the first part of (a) to retrieve all tuples whose value is greater than or equal to "Brooklyn" using the index, and then apply the additional criteria of assets $\geq$ 5000 on every tuple retrieved.

5. **Estimating Cost of Joins** (5 + 5 + 10 points)

Let relations $R_1(A, B, C)$ and $R_2(C, D, E)$ have the following properties: R_1 has 20,000 tuples, R_2 has 45,000 tuples. 25 tuples of R_1 fit on one block and 30 tuples of R_2 fit on one block. Assume that the join can not be all done in main memory, so it requires disk access. Estimate the number of block transfers and seeks required using each of the following join strategies for $R_1 \bowtie R_2$, assuming that there are M blocks of memory:

(a) Nested-loop join

(b) Merge join (assume neither are sorted on the join key but the tuples with the same bucket fit in memory)

(c) Hash join (assume no overflow occurs and calculate for both recursive partitioning and without recursive partioning)

(**ANSWER**)

R_1 needs 800 blocks and R_2 needs 1500 blocks (based on the size).

(a) Nested-loop join: If we use R_1 as the outer relation, we will need $20,000 \times 1,500 + 800 = 30,000,800$ disk accesses. If we use R_2 as the outer relation, we need $45,000 \times 800 + 1,500 = 36,001,500$ disk accesses.

(b) Merge join: Assume that R_1 and R_2 are not sorted on the join key, so the total cost of joining the two tables is: $B_s = 2 \times 1500(\lceil \log_{M-1} 1500/M \rceil + 1) + 2 \times 800(\lceil \log_{M-1} 800/M \rceil + 1)$ disk accesses. Assuming all tuples with the same value for the join attributes fit in memory, the total cost is $B_s + 1500 + 800$ disk accesses. Note that the slides were incorrect and off by a factor, but we accepted answers that had $B_s = 1500(2\lceil \log_{M-1} 1500/M \rceil + 1) + 800(2\lceil \log_{M-1} 800/M \rceil + 1)$.

(c) Hash join: Since R_1 is smaller, we will use that as the build input and R_2 as the probe input. The cost is then $4(1500 + 800) = 6900$ disk access. If recursive partitioning is used, the cost is $2(1500 + 1800)\lceil \log_{M-1} 800 - 1 \rceil + 1500 + 800$.

6. **Efficient Join Strategy** (15 points)

Consider the relations $R_1(A, B, C)$, $R_2(C, D, E)$, and $R_3(E, F)$ with primary keys A, C, and E, respectively. Assume that R_1 has 1000 tuples, R_2 has 1500 tuples, and R_3 has 750 tuples. Estimate the size of $R_1 * R_2 * R_3$ and give an efficient strategy (e.g., are there other indexes we should create in the relations, and what are the steps) for computing the join.

(**ANSWER**)

The relation resulting from the join of the three relations will be the same no matter which way we join due to associative and commutative properties of joins. Joining R_1 and R_2 will yield a relation of at most 1000 tuples since C is a key of R_2, since R_2 has 1500 tuples, each tuple has a unique c value, which means that for every tuple in R_1 it will at most match to 1 tuple in R_2. Likewise, joining that result with R_3 will yield a relation of at most 1000 tuples as well, since E is a primary key of R_3. Therefore, the final relation will have most 1000 tuples.

Since SQL creates an index for the primary keys, we do not need to create an additional indexes to speed up the process. The join process would then involve for each tuple, r_1 in R_1 we do the following:

- Use the primary index on C in R_2 to look up the tuple, r_2, which matches the C value of r_1.
- Use the primary index on E in R_3 to look up the tuple which matches the unique value of E in r_2.