Problem Set 2 Solutions

The problem set was worth a total of 25 points. Points are shown in parentheses before the problem.

Part 1 - Warmup (5 points total)

1. We studied three disk based 'access methods' in class – heap files, B+trees, and hash files built via extensible hashing. Suppose you have a table A with $|A|$ records in it and a table $|B|$ with B records in it, where $|A| < |B|$. Each table has fields, f_1 and f_2. For each of the following query workloads, describe which access methods would be best suited to that workload, including whether any indices should be clustered (primary) or unclustered (secondary). You may assume that sequential disk I/O is ten-times more efficient than random I/O. Justify each answer with a short sentence or analytical argument. You may disregard insertion and deletion costs.

   (a) A sequential scan of all elements of $|A|$.

   The best choice of access method is a heap file on A, since the pages need to be scanned in no particular order. Although other clustered access methods can also be used to scan relatively efficiently, pages are usually not arrayed sequentially and there is some space overhead due to key values and / or pointers.

   (b) A join of $|A|$ and $|B|$ of the form $(A.f_1 = B.f_1)$.

   There are several roughly equivalent ways to answer this question. A merge join on two sorted heap files, or between two B+-trees would likely be most efficient, since we are unlikely to have to read in any page of either relation more than once (the only way this could happen is if there are many – more than will fit in memory – duplicates of one particular value.) Alternatively, we could build a hash file on $B.f_1$ and a heap file on A, and perform an index nested loops join with B as the inner relation, which would be efficient if most or all pages of the hash file fit into memory.

   (c) 50% range queries of the form $(A.f_1 > k)$ on A (where k is a constant), and 50% joins of the form $(A.f_1 = B.f_1)$.

   By building a clustered B-tree on $A.f_1$, we can efficiently answer the range queries. For the joins, we can use a simple heap file on B and perform index nested loops joins with A as the inner relation. Alternatively, we could use a clustered B-tree or a clustered hash file on $B.f_2$ and perform merge joins or a nested loops join with B as the inner relation, as in the previous question.

   (d) 90% range queries of the form $(A.f_1 > k)$ on A, and 10% joins of the form $(A.f_2 = B.f_2)$.

   A clustered B+-tree on $A.f_1$ will allow us to answer range queries efficiently. With a B+-tree or hash file on $B.f_2$, we can perform an index nested loops join with B as the inner, using the link-pointers in the B+-tree on A to sequentially scan it as the outer. Note that in this approach we aren't accessing $A.f_2$ in sorted order, meaning that we may have to read some pages of B multiple times (unless the entire relation fits in memory) and we won't access B in sorted order. Assuming that we are allowed only one clustered index per relation, the other viable alternative is to keep a secondary $B + tree$ on $A.f_2$, and perform a merge join – this requires random access in to the pages of A, which will be inefficient as well.

   (e) 33% joins of the form $A.f_1 > B.f_1$, 33% sequential scans of B, and 33% lookups of the form $B.f_2 = k$.

   We can use a clustered B+-tree on $A.f_1$ and $B.f_1$ to compute the joins and sequential scans reasonably efficiently. An secondary hash index on $B.f_2$ will allow us to compute the lookups quickly.

Part 2

Bob is a vacuum cleaner salesman. He has decided to store information about his clients and sales in a database. He wants to keep the following information:

(a) For each customer, he wants to store his or her name, address, and the sales price of all the vacuums he or she has purchased (some customers own multiple vacuums, and one crazy old man even own three of the same type of vacuum!) Bob sells his vacuums for different prices to different customers.

(b) Each vacuum model has a different horsepower rating which Bob also wants to store.

(c) As a way to increase his profit margin, Bob tries to get customers to buy a one-year 'service contract' with the vacuums he sells. Bob visits customers who have bought a service contract and maintains their vacuum for them. Thus, for each sale, Bob wants to keep track of the sales price and the type of service contract (if any) attached to the sale.

(d) Each type of service contract has a service frequency associated with it, which Bob wants to store in the database.

(e) For all service contracts, Bob wants to store the last time the customer's vacuum was serviced.

Bob, knowing he's not being the brightest bulb on the tree, is concerned that he won't be able to write complex queries that combine readings from multiple tables together to get information about his customers and their vacuums. Thus, he proposes to store his database in a single table, reasoning that this way he only has to store one record per vacuum he sells, and it will be very easy for him to lookup information about his business. His schema is as follows:

vacuums: {cid | name | city | street | no | model_no | hp | price | contract_type | svc_freq | last_svc }

This representation works fine for a few months, but as Bob's customer base grows, he finds it is very hard to maintain this database, particularly as customer's retire their old vacuums in favor of new models with new service contracts. He hires you to help him fix his database system.

2. (1 point) Derive a set of non-trivial functional dependencies for this database from Bob's requirements given above.

   Assuming that no customer buys the same model of vacuum twice, reasonable functional dependencies would be:

```
cid -> name, city, street, no
contract_type -> svc_freq
model_no -> hp
cid,model_no -> price, contract_type, last_svc
```

   In reality, to make this a good schema, and support the possibility that a particular customer might buy the same kind of vacuum twice, we should introduce a transaction id key for each purchase, which would change the last dependency above to something like:

```
tid -> cid,model_no, price, contract_type, last_svc
```

3. (1 point) List three update, insertion, or deletion anomalies that Bob might encounter using the schema he has proposed.

   - Every time a particular model of vacuum is purchased, Bob must enter the horsepower information; if he does this incorrectly, the same model of vacuum can be listed with two different horsepowers.
   - Every time a contract of a particular type is purchased, the service frequency is entered in the new record; he could do this incorrectly, causing two records with the same service contract type to have different service frequencies.
   - When a customer changes his or her address (or name), Bob might have to change that address (name) in multiple records. If he doesn't do this, there will be an update anomaly.

   There are several possible variants on these basic anomalies.

4. (1 point) Use your functional dependencies to derive a schema in BCNF for Bob's database.

   Assuming the use of the tid field above:

```
customer : {cid | name | city | street | no}
contracts : {contract_type | svc_freq}
vacuums : {model_no | hp}
sales : {tid | cid | model_no | price | contract_type | last_svc}
```

   Does your schema preserve all of the functional dependencies you began with? Does it address all of the anomalies you listed above?

   Yes, and yes.

5. (3 points) Suppose that Bob wants to find the name's and addresses of all of the customer's who live on Main St. whose vacuums are due for service. Suppose that 75% of customers own service contracts, 25% of them live on Main St, and 10% of the service contracts (irrespective of contract type) are due for service. Assume there is no relationship between where a customer lives, the likelihood that he or she purchases a service contract, or the likelihood that a particular contract is due for service. Write a SQL query that accomplishes this for both Bob's original schema and for the schema you designed. Show equivalent relational algebra operations for these two queries, and argue that the algebra you have selected is an efficient way to execute your query given the operator selectivities above.

   For Bob's schema:

```
SELECT name, address
FROM vacuums
WHERE street = 'Main'
AND last_svc + svc_freq > now()
AND contract_type != no_contract
```

   For our schema:

```
SELECT name, address
FROM customer, contracts, sales
WHERE contracts.contract_type = sales.contract_type
AND customer.cid = sales.cid
AND customer.street = 'MAIN'
AND sales.last_svc + contracts.svc_freq > now()
AND customer.contract_type != no_contract
```

   Figure 1 shows the equivalent algebra for these two queries. Basically, we need to push down selections as far as they can go, project away unused fields, and order the joins appropriately. In this case, we're better doing the join between sales and contracts first because it is more selective, and the cardinality of the join output is smaller (it's also likely to be very fast as the contracts relation is small so we can keep it in memory.)

6. (5 points) For each of the operators in your two query plans, describe the particular implementation (i.e., physical plan) you would choose given the parameters below. You should consider issues such as the appropriate choice of join algorithm, which relation is inner or outer, and so on. You should compute the approximate number of I/Os for each of the two plans.

   Assume that all of your tables are stored in a heap file with no indices. You may also need to know that Bob has sold 10,000,000 vacuums to 8,000,000 customers under 10 different types of contracts. Assume that names, cities, and streets each require 20 bytes to store, and that all other record types (including `null` values) require 2 bytes to store. You may assume that there is no additional per-tuple storage overhead. Bob's computer has 1MB of memory for query processing, and a disk page is 10KB.

   Physical memory is $10^6/10^4 = 100$ pages.

   For Bob's schema, there are no join operators, so the physical plan looks very much like the logical relational algebra shown in Figure 1(a). The size of a single record in his schema is $8 \times 2 + 3 \times 20 = 76\ bytes$. The projection does not save us any I/Os. Since this is a fully pipelined query plan, the cost of this approach in terms of number of I/Os then simply the cost to read the whole relation off disk, e.g., $10^7 \times 76\ bytes/10^4\ bytes/page = 7.6 \times 10^4 = 76,000\ pages$. We will assume we can directly output this result to the user (e.g., that there is no materialization at the top of the plan.)

   For our schema, we begin by considering the cost of joins. The first join, between `contracts` and `sales`, should be done as a nested loops join, with `contracts` (which can fit in memory) as the inner relation. This join can be done with no additional I/Os beyond those required to read the relations off of the disk.

   For the second join, neither input will fit into memory. The most efficient way to do this is via a so-called "block nested loops join", which works as follows: assume the outer relation of the loop is the `cid` field of the filtered `sales/contracts` relation, which is $2 \times 10^7 bytes \times .075/10^4\ bytes\ per\ page = 150\ pages$. There then are $8 \times 10^6 \times .25 = 2,000,000\ customers \times 64\ bytes/customer/10^4 bytes\ per\ page = 12,800\ pages$ in the inner

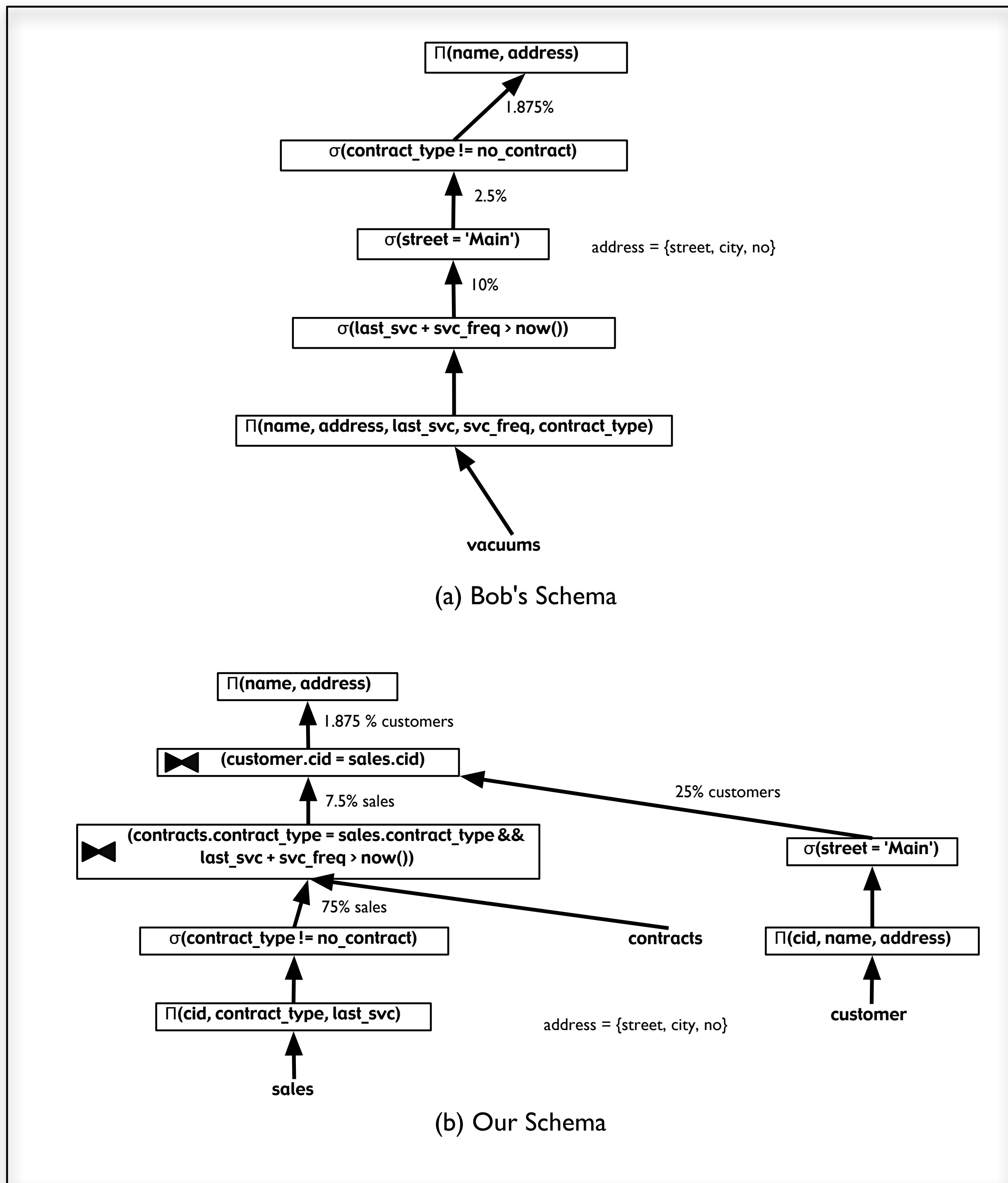

Figure 1: Relational algebra for Bob's schema (a) and our schema (b).

relation. We can read in the first 75 pages of the outer, and scan the inner, and then rescan with the second 75 pages of the outer. This requires us to read a total of $150 + 2 \times 1.28 \times 10^4 = 25,750\ pages$. Hash-join and sort-merge join can also do this in approximately two passes over the larger of the two relations (one pass to write out sorted runs, and another to read it back in sorted order.)

We also need to consider the costs of reading the base relations, and of materializing the inputs to the second join. The `customer` relation is $8 \times 10^6 \times 64\ bytes/10^4\ bytes/page = 51,200\ pages$ that must be read off disk. The sales relation consists 12 bytes per record, for a total of $10^7 \times 12/10^4 = 12,000\ pages$ that must be read. The `contracts` relation requires 1 page to read. The cost of writing the filtered `customer` relation to disk (*materializing* it) is 12,800 pages, and the cost of materializing the joined `sales/contracts` relation is 150 pages (as above).

Thus, the total number of I/Os is approximately 51,200 + 12,000 + 12,800 + 150 + 25,750 = 101,900. It's possible to reduce this by 12,800 if you allow the first 75 tuples of the `sales/contracts` materialized result to be joined with the 12,800 records of the materialized `contracts` relation as the `contracts` relation is written to disk. In this case, the total cost would be about 89,100 I/Os.

7. (4 points) Suppose that you are allowed to add primary and/or secondary indices to the tables in your schema (you do not need to consider Bob's schema in this question.) Describe what indices you would add, and recompute the overall number of I/Os when these indices are available. You may assume that selections and join predicates can be pushed into access methods via SARGs (e.g., that if you have a hash file on street name, you can directly look up customers who live on Main St.)

   If we build a primary B+-tree index on `sales` in order of first `contract_type` and then `cid`, we can use a SARG to read just 75% of the sales records from disk. Because there are few values of contracts, most of the records in the B+-tree can be scanned using the sibling pointers – only when we encounter a page that has value `no_contract` do we have to read intermediate pages in the index to look up the first record with the next `contract_type` value. Note that after we filter by `contract_type`, the remaining records are in `cid` order.

   If we then build a primary B+-tree on `customer` with records sorted by `street` followed by `cid`, we can use a SARG on street name to read just the 12,800 pages of the relation from disk (since we only have to read consecutive pages of customers living on Main St.) Those records are then in `cid` order, and we can perform a sort-merge join for the second join. This allows us to avoid having to write any intermediate results to disk, since both inputs are already in sorted order (this might not be the case if there are long runs of duplicate `cid`s in the `sales` records – which is unlikely.) Thus, the total number of I/Os is about $12,800 + .75 \times 12,000 = 15,800\ I/Os$.

   An alternative, but less efficient approach, is to build a primary B+-tree index on just the `cid` field of the `customer` table so that we read customers off disk in `cid` order. We can use the join predicate `(customer.cid = sales.cid)` as a SARG on `customer` (assuming `customer` is the inner relation), allowing us to read only 7.5% (the percentage of `sales` records that are output by the first join) of the base records of the `customer` relation. Of course, this means we effectively push the predicate on `street` until after the join, but this doesn't affect the total number of I/Os. Unfortunately, the 7.5% of the records that are selected are not consecutive. If we assume they are uniformly distributed, we will have to read 1/.075 = 1 in 15 records. Since there are $10000\ bytes\ per\ page/64\ bytes\ per\ record = 156\ records\ per\ page$, the probability that every page will have one matching record on it is very high, suggesting we will still have to scan all 51,200 records of the `customers` relation.

   Thus, this alternative approach requires approximately $51,200 + 12,000 \times .75 = 60,200\ I/Os$.

Part 3

8. (5 points) In a question unrelated to Bob's vacuum business, suppose that you are designing a database system that stores each column of data in a table in a separate file on disk. Such a representation can be efficient when, for example, most queries access only a few columns but each table consists of tens or hundreds of columns, since each query must then access only the disk pages corresponding to columns it references. When a representation of this sort is used, the database system needs a facility figure out which records of a given column correspond to records in another column. If all columns are stored in the same order, this correspondence is trivial; however, we may wish to store different columns

in different orders to facilitate common queries over columns in a particular order. For example, if we store a `salary` column in sorted order, it is very easy to look up employees who make more than $50,000.

In the situation where columns are stored in different sort orders, we can store a special column, $P_{A->B}$, of values called a "permutation", that maps the ith entry in a column A to the $P_{A->B}(i)$th entry in B. With $n-1$ (where n is the number of columns in the original table) such permutations, we can reconstruct the original table.

Your job is to devise an efficient external (i.e., disk-based) algorithm for permuting a column A into the order of a column B using a permutation $P_{A->B}$. Assume that you have M bytes of memory, and that A and P each require n bytes to store and consist of r records. Assume that $n >> M$ and that a disk pages are B bytes. Characterize the performance of your algorithm in terms of the number of sequential and random disk I/Os (i.e., page reads and page writes) it requires.

An example of the permute operation is shown below:

```
permute(A={a,b,c,d,e,f,g},  P_{A->B}={2,1,4,5,6,3,7}) =
P[1] = 2;  A[2] = b
P[2] = 1;  A[1] = a
P[3] = 4;  A[4] = d
P[4] = 5;  A[5] = e
P[5] = 6;  A[6] = f
P[6] = 3;  A[3] = c
P[7] = 7;  A[7] = g
=> b,a,d,e,f,c,g
```

Each page of A and P contain Br/n records. There are a total of n/B pages in each. We can hold M/B pages in memory.

Our algorithm works as follows. We begin by reading the first $(M/2B)-1$ pages of records in A as well as the first $(M/2B)-1$ pages of records in P. Note that these sets of pages contain the same number of records, and that the ith element in our pages from P tells us where the ith element in our pages from A should go in B. We can then sort the pages of P in place, using, e.g., an insertion sort, performing the same swaps on A as we do on P. We can then write out a list of pairs of the form $(P[i], A[i])$ sorted in $P[i]$ order. We then repeat this process with the remaining $\lceil (2n/M)-1 \rceil$ sets of $(M/2B)-1$ pages in A and P. This phase consists entirely of sequential I/O to read and write pages from disk, and we have to read and write both A and P once, for a total of $4n/B$ sequential I/Os.

At the end of this process, we have $\lceil 2n/M \rceil = k$ sorted runs of pages. If $k < M/B$, we can trivially complete the permutation by reading one page from each of the runs and performing a k-way merge sort, writing out the final pages of B as we go.

If $k \geq m$, then we need reduce the number of sorted runs, which we will do by recursively applying M/B-way merges to the sorted runs – e.g., we will read from M/B runs at a time and merge sort, producing new runs of length $M^2/4B^2$. If we still have more than M/B runs at this point, we will recurse again. Thus, we reduce the number of runs by M/B at each iteration, so we have to a total of $\lceil log_{M/B}(n/2B) \rceil$ passes, reading and writing all of the data on each pass, for a total number of I/Os of $4n/B \times \lceil log_{M/B}(n/2B) \rceil\ I/Os$. Of these, the first $4n/B$ are sequential, and the remainder are random as we read or write one page at a time from or to each of the sorted runs. We can convert the remainder of I/Os to be sequential by reading and writing larger blocks from fewer of the runs at a time, which will decrease the base of the logarithm, but will improve I/O performance. It's difficult to evaluate this tradeoff without specific numbers for the various sizes and the ratio of cost of sequential to random I/O.

(Note that this is the same cost as an external sort – the fact that we have the permutation doesn't help us!)