6.830 Fall 05, PS2 Solutions
10/19/2005

1) [3 points] The following is one possible schema for the pet store database:

customers: (<u>cid</u>, name, address, cc#)
animals: (<u>aid</u>, aname, species, breed, size, *ownerid*)
orders: (<u>ordid</u>, *cid*, orderdate, shipdate)
products: (<u>pid</u>, type, species, color, price, quantity_stocked)
order_info: (<u>*ordid*</u>, <u>*pid*</u>, quantity)

Primary keys are underlined. Foreign keys are in italics, and are as follows:

animals.ownerid references customer.cid
orders.cid references customers.cid
order_info.pid references products.pid
order_info.ordid references orders.ordid

Functional dependencies are as follows:

cid -> name, addr, cc#
aid -> aname, species, breed, size, cid
orders -> ordid, cid, orderdate, shipdate
products -> pid, type, species, color, price, quantity_stocked
order_info -> ordid, pid, quantity

This is in BCNF since only primary keys appear on the left hand side of any functional dependency.

2) [3 points] This question is somewhat ambiguous. The queries shown here compute all orders that contain a product that is for a species of pet that the owner customer placing the order does not own. They report the name of the customer, the order id, and the type of the product that is for a type of pet the customer does not own.

The straightforward way to compute this query is using a correlated subquery (e.g., a subquery that references an expression in the outer query):

```
SELECT DISTINCT c.name, o.ordid, p.type, p.species
FROM orders AS o, customers AS c, products AS p, order_info AS inf, animals AS a
WHERE c.cid = o.cid
AND inf.pid = p.pid
AND inf.ordid = o.ordid
AND p.species NOT IN
(SELECT DISTINCT species AS s1  --all the species this customer owns
     FROM animals
     WHERE c.cid = animals.ownerid);
```

We can get rid of the correlation in the subquery, as follows:

```
SELECT DISTINCT c.name, o.ordid, p.type, p.species
FROM orders AS o, customers AS c, products AS p, order_info AS inf, animals AS a
WHERE c.cid = o.cid
AND inf.pid = p.pid
AND inf.ordid = o.ordid
AND (c.cid,p.species) NOT IN
(SELECT DISTINCT customers.cid, species AS s1
        FROM animals, customers  WHERE customers.cid = animals.ownerid);
```

We'll assume the second query is the one to use.

Note that when the query is re-written in this way, it's clear we don't actually need the reference to customers in the inner query at all:

```
SELECT DISTINCT c.name, o.ordid, p.type, p.species
FROM orders AS o, customers AS c, products AS p, order_info AS inf, animals AS a
WHERE c.cid = o.cid
AND inf.pid = p.pid
AND inf.ordid = o.ordid
AND (c.cid,p.species) NOT IN
(SELECT DISTINCT ownerid, species AS s1
        FROM animals);
```

3) [3 points] When there is abundant memory, indices don't help. Look at the plan in Figure 1 (which is the solution to problem 4.) Each of these joins is a key-foreign key join, and hence produces an output equal to the size of the larger (primary key table.) In some cases, we could use a primary index on the inner relation to efficiently lookup join results, but in general we would have to do a very large number of index lookups, since each tuple in the inner relation always joins with at least one tuple in the outer relation. These index lookups would tend to be random. To avoid this random I/O, a better choice would be hash or merge joins with hash-tables built or resorting done on the fly. With sufficient memory, these data structures allow us to sequentially scan each inner relation just once in sequential order. No indices are needed.

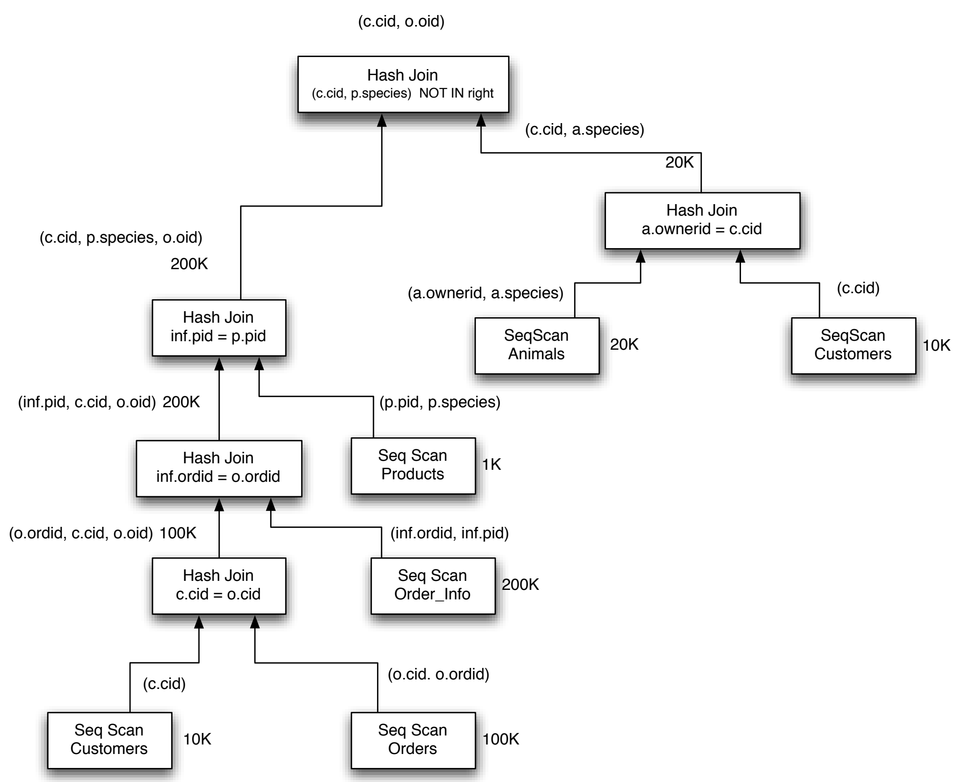

Figure 1: Possible plan for second query given in 2).

4) [3 points] See Figure 1. There are only three join orders of interest for the left subquery-- we could start by joining products to order_info, orders to order_info, or customers to orders. The correct choice is the ordering that minimizes the number of tuples along each arc. Since any join that starts with order_info would immediately put 200K tuples along each arc, we should start with the customers/orders join, which only produces 100K tuples. We continue in a left-deep fashion since that allows us to pipeline the outer relation (without hashing/partitioning it, since we have sufficient memory to hold the hash on the inner in RAM), arriving at the join ordering shown here. There is only one possible ordering for the right plan

5) [3 points] As noted above, the plan shown here has to sequentially scan each relation once. There are 10 tuples per page. Hence, we have to perform the following I/Os:

customers: 10K / 10 = 1K I/Os
orders: 100K / 10 = 10K I/Os
pets: 20K / 10 = 2K I/Os
products: 1K / 10 = 100 I/Os
order_info: 200K / 10 = 20K I/S
total: 33,100K sequential I/Os

6) [5 points] 1MB of memory is enough to hold approximately 1000 pages of data. We have four join operators -- let's assume we divvy these pages up evenly between them; hence, there are 250 pages of memory per operator. For all of our joins except products with order_info (with products as the inner), we will need to use an external join that writes some data to disk. Recall from Shapiro that as long as we have sqrt (relation size) pages of memory, we can compute an external hash join in at most three sequential passes over the data (one to read, one to write out hash partitions, and one to process each of the hash partitions). The largest relation is 20,000 pages; 20,000 ^ 1/2 = 141, so we have plenty of memory per join to do this. In addition to scanning each of the inner relations three times, we will also have to write and read the outer intermediate results several times, since we have to partition them before joining. Hence, the total I/Os will be a small constant factor larger than the number with unlimited memory. All I/Os will still be sequential (note that while writing out hash-runs in the external hash, we will have to seek between disk writes, but that we get to write a page of tuples at a time, rather than seeking between every tuple. So this is not pure random I/O, but is not purely sequential either. We count it as sequential I/Os below.)

Other plans with hash joins will perform no better. Plans with index nested loops would be unlikely to perform better, since they still perform random I/Os and, with large relations (such as order_info), only a small fraction of the pages will fit into the buffer pool, requiring many pages to be re-read multiple times.

We can implement the NOT IN operator using a partitioned external hash, just as with the external join operator.

Hence, the total number of I/Os is (we assume for these calculations that intermediate results still occupy 100 bytes per tuple, even though projections will reduce these tuple sizes somewhat):

33,100 +
2 (1K) + //pages of I/O from read and write phases of external hash on customers
2 (10K) + //R/W phase of external hash on orders
2 (10K) + //R/W phase of external hash on intermediate result of 1st join
2 (20K) + //R/W phase of external hash on order_info
  // **Note** that we can skip external hash on results of 2nd join, since hash on
  // products fits into memory
2 (20K) + //R/W phase of external hash on intermediate result of 3rd join
2 (2K) + //R/W phase of external hash on result of subquery

2 (2K) + //R/W phase of external hash on animals in subquery
2 (1K) + //R/W phase of external hash on customers in subquery

33,100 + 132,000 = 165,100 I/Os, (pseudo) sequential