Polytechnic Institute of NYU
Computer and Information Science
CS6083, Spring 2011

Solution for Query Processing Example

Q1: “List the names of all customers who rented Rush Hour 3 on November 22, 2007”
Q2: “For any rental of an action movie released in 2007, list the title and customer name”
Q3: “List the title of any movie that was rented at least once within 180 days of its release”
Q4: “For any rental of an action movie released in 2007 by a customer in Oakville, list the movie title and customer name”

(b) Assuming there are no indexes, we have to scan each relation that is involved at least once, or several times if both relations to be joined do not fit into main memory.

Q1: Neither relation fits completely into main memory. However, we can scan Rental once and only keep those records that concern a rental of Rush Hour 3 on November 22, 2007. There are about 5000 rentals of Rush Hour 3 if we assume that this movie was rented as frequently as an average movie (5000 = 2 billion /400,000), and 2% of the rentals (100) were on that date. So after scanning Rental, we have only 100 records left, which fit in main memory. Thus, we only have to scan Customer once to find the matching customer names. So the cost is the same as that for scanning the Customer table (2 GB) and the Rental table (160 GB), which takes time 162,000 MB / 40 MB/s = 4050 seconds (ignoring the 16ms for two seeks to find the start of each file, which is negligible).

Q2: We first scan the Movie relation once, keeping only those records that are about action movies released in 2007. There are about 4000 such movies (20% of 5% of 400,000 movies total), so these fit into main memory. We scan Rental once to join these records. Since each movie on average has been rented 5000 times, this means we might have 20 million records after this join, where the size of each record is 280 bytes (200 bytes for each Movie record, and 80 for each Rental record). Thus, these records use 5.6 GB, and would not fit into main memory for the join with customer.

Of course, we can project away everything from the resulting records except the customer ID and movie title, so realistically the size per record might be only, say, about 48 bytes (8 bytes for customer ID, and maybe 40 for movie title), or 960 MB total. In this case, we would only scan the customer table 5 times (since 5 * 200MB > 960MB). So total cost is cost of scanning Movie and Rental once and Customer 5 times, and we also need to write and then read again the 960MB of intermediate results (but not really; see below). Thus, cost is (80 + 160,000 + 2 * 960 + 5 * 2,000) / 40 = 172,000 / 40 = 4300 seconds. If we do not project away the unnecessary attributes, then we would have to read Customer 28 times (since 28 * 200 MB = 5.6 GB), and write and read 5.6 GB of intermediate results, so the cost would be (80 + 160,000 + 2 * 5600 + 28 * 2,000) / 40 = 227,280 / 40 = 5682 seconds. We ignore the cost for writing the final output, which is the same for all solutions.

A few remarks about this query, which was a little tricky: First, as we saw, projecting redundant attributes out of the intermediate results early can speed up processing significantly. Second, note that we actually do not have to write the intermediate 960MB or 5.6 GB of data out to disk and read them back in: during the join between Movie and Rental, we keep the 4000 Movie records in a small fraction of the available main memory, and scan the Rental table once, producing matches (joined records) while we do the scan. Whenever we have produced enough temporary records to fill up the available memory, we then perform one complete scan of the Customer table to look for matches. Thus, the two join operations could be overlapped (pipelined), as described in the context of Volcano query processing, resulting in less disk traffic. Finally, if it is really important to keep the intermediate result set small, there are some additional tricks (apart from early projection) that can be used. For example, while every record from Movie joins with 5000 records from Rental, it might not really be necessary to replicate all the information in each Movie record 5000 times – it might be smarter to just have a pointer to that record instead. BTW, you are not expected to do all of this for full credit in the homework or exam -- if you observed that in the second query, as opposed to the first one, the intermediate result does not fit and thus several scans of Customer are needed in a block-based nested-loop join, then that is enough.

Q3: The tricky issue with this query is that the selection condition involves both release date and rented date, so it can only be checked after the join of Movie and Rental. Luckily, Movie has a size of only 80 MB and thus fits into memory, so we only need to scan each relation once. Cost is (80 + 160,000) / 40 = 160,080 / 40 = 4002 seconds.

Q4: Note that this query is almost identical to the second one, except that we have an additional condition on the city where the customer lives. This means we can execute this query either from the left (starting with a selection on Customer) or from the right (starting with the selection on Movie). We already saw in the second query that starting from the right results in several scans of the Customer table, because the intermediate results after joining with Rental do not fit in main memory. So how about if we start from the left? After scanning Customer once, we have 20 tuples of people living in Oakville. Assume that people in Oakville are like average customers, so they have done about 200 * 20 = 4,000 rentals. Thus, after joining the 20 Customer tuples with Rental, by scanning Rental once, we have 4,000 tuples that need to be joined with Movie. These 4,000 tuples fit into main memory, so we only have to scan Movie once. (In fact, since Movie itself fits in main memory, this is not even crucial.) Overall, it is better to process this query starting with the selection on Customer, and we end up scanning each table exactly once, for a cost of (2,000 + 160,000 + 80)/40 = 162,080/40 = 4052 seconds.

Overall, in all these queries the main bottleneck is that without any index, the largest table, Rental, has to be completely scanned at least once.

(c) Each tree node contains n-1 keys and n pointers. With 12 bytes per key, 8 bytes per pointer and a node of total size $4 * 2^{10} = 4096$ bytes, we can find how many keys and pointers can fit in each node:

$$(n-1) * 12 + 8 * n = 4096 \quad => n = 205$$

Assuming 80% occupancy per node, each leaf node will contain about 164 index entries, and each internal node will have about 164 children (but you can round these numbers as convenient and make your own assumptions about the average occupancy).

For the Customer table, 20 records fit into each disk page, assuming 100% occupancy in the disk blocks used by the relation. Thus, a sparse index on Customer will only have one index entry for every 20 records, or a total of 500,000 index entries. Since about 164 index entries are in each leaf node, there will be about 500,000 / 164 = 3049 leaf nodes. On the next level, there will be about 3049/160 = 19 nodes, and then the next level is the root of the tree. So the B+-tree has 3 levels of nodes: the root, one internal level, and the leaf level. Thus, it takes about 4*8ms = 32ms to fetch a single record from the table using this index, assuming no caching. The size of the index is dominated by the leaf level, which is about 3049 * 4 KB = 12.4 MB.

For the dense index on Rental, there will be 2 billion index entries, in 2,000,000,000 / 160 = 12,195,122 leaf nodes, and 12,195,122 / 164 = 74,360 on the next level, then 453 entries, then 3, and then the root. So this tree has 5 levels of nodes, and 6 * 8ms = 48 ms are needed to fetch a single record from the table (assuming no caching). The size of the index is again mainly determined by the leaf level, which is 12,195,122 * 4 KB, or about 50 GB.

(d) For the first query, the index on the date_rented attributed of Rental can be used to fetch those movies that were rented on November 22, 2007. (There is no index on the title attribute of Rental, only on the title attribute of Movie.) 0.2% of all rentals were done on that particular date, resulting in a total of 4 million Rental records. Fortunately, this is a clustered index, so we can efficiently scan all these records, which take up 4 million * 80 bytes = 320 MB. So fetching these records takes about 320 MB / 40 MB/s = 8 seconds, and this time dominates the small amount of time needed to actually traverse the index from top to bottom once. After scanning these 320 MB, we only keep those records that are about Rush Hour 3, which are about 100 records, or 8 KB of data. To do the join with the Customer table, we cannot use an index (since Customer has no index on cid), but we can do a block-nested loop join with one simple scan of the Customer table, at a cost of about 2,000 / 40 = 50 seconds. So the total cost of the first query now reduces to about 58 seconds.

For the second query, we can use either the index on genre or on date_released of the Movie table, but not both. 5% of all movies are action movies, but only 2% were released in 2007, so we use the index on date_released. This gives us 8000 Movie records, and for each record we have to do one random lookup of about 8 ms. So this would take about 64 seconds, but a simple scan of the Movie table would take only 2 seconds, so it is better to not use any index on Movie. (This would be different if we had a clustered index on Movie.) Note that as a result, we get 4000 records from the Movie table concerning action movies released in 2007. For the Rental table, there is no index we can use for the join with Movie, so the cost stays the same (one scan of the Rental table), and for the Customer table there is also no improvements, since we have 20 million records from the join of Movie and Rental that need to the joined with Customer, so using an index would be a very bad idea. Overall, there is no benefit in using the given indexes for this query.

For the third query, there is also no use for the given indexes. The selection can only be done after the join, so indexes are not useful in this case. For the join itself, if there was an index on title of Rental, we could then do an index-based join, resulting in one lookup into Rental for each tuple in Movie. This would be 400,000 lookups of 48ms each in the case of a clustered index, or about 19,200 seconds, but this is slower than a full scan of Rental (4,000 seconds), so this is not a good idea.

(e) Here is what we can do to improve the running time of the second query: Having a composite clustered index on the (genre, date_released) attributes of Movie would speed up this first part from 2 seconds to maybe about 4*8 + 20 = 52ms since the 4000 action movies released in 2007 would be in one 800 KB contiguous chunk of the Movie relation (assuming an index of height 3). If we then have a clustered index on title in Rental, we could perform 4000 lookups on movie title, since for each title the Rental records would be contiguous in Rental. Assuming a height of 4 for this index, each lookup would take 5 * 8 ms = 40 ms, and totally this would be 160 seconds, much better than before (and even better once you realize that parts of the index structure will be cached in main memory). We would then again get 20 million tuples that need to be joined with customer, and the cost for this join stays the same, about 250 seconds, so total cost is about 410 seconds, versus several thousand seconds before.

(f) Now consider Q4 and what indexes to build for it. Recall that this query was similar to Q2, but we found that without index structures, it was better to start with the Customer table rather than the Movie table (as in Q2). This is also true for the indexed case. We would choose a clustered index on city in Customer so we can efficiently fetch the 20 cids of all customers living in Oakville. These 20 tuples would take up about 4KB of contiguous space, so fetching this via an index of height 3 would take about 4*8 = 32ms (this is dominated by seek cost, so we ignore the transfer cost for 4KB). Then we could use a clustered index on cid in Rental to perform 20 lookups on the cids, which would take about 20 * (5 * 8) = 800ms, or 0.8 seconds, for an index of height 4. This results in 4000 tuples that we need to join with the Movie table. If we had an index on title in Movie, we could use this to do the join, by performing 4000 lookups. But note that scanning the entire Movie table only takes 2 seconds, so the index would be slower. Overall, the cost is now about 2.832 seconds.

Now suppose there are only 10 action movies released in 2007. In that case, there is an even better solution where we start from both sides. That is, we perform a selection on city in the Customer table using an appropriate index, and at the same time we also use an index on Movie to find action movies released in 2007. Now we attack the Rental table from both sides, using an unclustered index on cid to look up the records for the 20 customers in Oakville, and using an unclustered index on title to look up the records for the 10 action movies. However, in these lookups, we only go to the leaf level of the indexes and fetch the record IDs (RIDs) of the matching tuples. Then we can intersect the RIDs we fetch using the index on cid with the RIDs we fetch using the index on title, to get the final result. You can check and verify that this is in fact even faster. Of course, remember that in reality you cannot create all indexes that are optimal for some query, but you can choose just a few.