Computer Science 143, Homework 3

Michael Wu
UID: 404751542

May 22nd, 2018

Part 1

I found all the distinct company values and was able to generate the following query to convert from a long format to a wide format. This gave a result with 47 columns and 67 rows.

```
SELECT company,
  MAX(IF(value='bonded-by-product', 1, 0)) AS bonded_by_product,
  MAX(IF(value='project-ownership', 1, 0)) AS project_ownership,
  MAX(IF(value='friends-outside-work', 1, 0)) AS friends_outside_work,
  MAX(IF(value='feedback', 1, 0)) AS feedback,
  MAX(IF(value='impressive-teammates', 1, 0)) AS impressive_teammates,
  MAX(IF(value='personal-growth', 1, 0)) AS personal_growth,
  MAX(IF(value='fast-paced', 1, 0)) AS fast_paced,
  MAX(IF(value='lunch-together', 1, 0)) AS lunch_together,
  MAX(IF(value='remote-ok', 1, 0)) AS remote_ok,
  MAX(IF(value='customer-first', 1, 0)) AS customer_first,
  MAX(IF(value='many-hats', 1, 0)) AS many_hats,
  MAX(IF(value='quality-code', 1, 0)) AS quality_code,
  MAX(IF(value='open-communication', 1, 0)) AS open_communication,
  MAX(IF(value='internal-promotion', 1, 0)) AS internal_promotion,
  MAX(IF(value='retention', 1, 0)) AS retention,
  MAX(IF(value='product-driven', 1, 0)) AS product_driven,
  MAX(IF(value='worklife-balance', 1, 0)) AS worklife_balance,
  MAX(IF(value='light-meetings', 1, 0)) AS light_meetings,
  MAX(IF(value='flex-hours', 1, 0)) AS flex_hours,
  MAX(IF(value='inclusive', 1, 0)) AS inclusive,
  MAX(IF(value='psychologically-safe', 1, 0)) AS psychologically_safe,
  MAX(IF(value='diverse-team', 1, 0)) AS diverse_team,
  MAX(IF(value='eq-iq', 1, 0)) AS eq_iq,
  MAX(IF(value='parents', 1, 0)) AS parents,
  MAX(IF(value='open-source', 1, 0)) AS open_source,
  MAX(IF(value='continuous-delivery', 1, 0)) AS continuous_delivery,
  MAX(IF(value='engages-community', 1, 0)) AS engages_community,
  MAX(IF(value='cross-dep', 1, 0)) AS cross_dep,
  MAX(IF(value='safe-env', 1, 0)) AS safe_env,
  MAX(IF(value='rapid-growth', 1, 0)) AS rapid_growth,
  MAX(IF(value='new-tech', 1, 0)) AS new_tech,
  MAX(IF(value='creative-innovative', 1, 0)) AS creative_innovative,
  MAX(IF(value='data-driven', 1, 0)) AS data_driven,
  MAX(IF(value='flat-organization', 1, 0)) AS flat_organization,
  MAX(IF(value='engineering-driven', 1, 0)) AS engineering_driven,
  MAX(IF(value='agile-dev', 1, 0)) AS agile_dev,
  MAX(IF(value='pair-programs', 1, 0)) AS pair_programs,
  MAX(IF(value='team-oriented', 1, 0)) AS team_oriented,
  MAX(IF(value='internal-mobility', 1, 0)) AS internal_mobility,
  MAX(IF(value='physical-wellness', 1, 0)) AS physical_wellness,
  MAX(IF(value='benefit-company', 1, 0)) AS benefit_company,
  MAX(IF(value='interns', 1, 0)) AS interns,
  MAX(IF(value='junior-devs', 1, 0)) AS junior_devs,
  MAX(IF(value='risk-taking', 1, 0)) AS risk_taking,
  MAX(IF(value='good-beer', 1, 0)) AS good_beer,
  MAX(IF(value='office-layout', 1, 0)) AS office_layout
FROM keyvalues
GROUP BY company;
```

Part 2

1. No Subquery with `DISTINCT`

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	extra
1	SIMPLE	caltrans	NULL	ALL	PRIMARY	NULL	NULL	NULL	103056	20.99	Using where; Using temporary; Using filesort

2. `SELECT` within a `SELECT`

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	extra
1	PRIMARY	<derived2>	NULL	ALL	NULL	NULL	NULL	NULL	2402	100.00	Using temporary; Using filesort
2	DERIVED	caltrans	NULL	ALL	PRIMARY	NULL	NULL	NULL	103056	2.33	Using where; Using temporary; Using filesort

3. `JOIN` as a Filter

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	extra
1	PRIMARY	<derived2>	NULL	ALL	NULL	NULL	NULL	NULL	24047	100.00	Using temporary; Using filesort
2	DERIVED	c	NULL	ALL	PRIMARY	NULL	NULL	NULL	103056	2.33	Using where; Using temporary; Using filesort
2	DERIVED	<derived3>	NULL	ref	<auto_key0>	<auto_key0>	8	hw2.c.highway	10	100.00	NULL
3	DERIVED	caltrans	NULL	ALL	PRIMARY	NULL	NULL	NULL	103056	2.33	Using where; Using temporary

4. Using an `IN` Subquery as a Filter

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	extra
1	PRIMARY	<derived2>	NULL	ALL	NULL	NULL	NULL	NULL	2402	100.00	Using temporary; Using filesort
2	DERIVED	c	NULL	ALL	PRIMARY	NULL	NULL	NULL	103056	2.33	Using where; Using temporary; Using filesort
2	DERIVED	<subquery3>	NULL	eq_ref	<auto_key>	<auto_key>	265	hw2.c.highway,hw2.c.area	1	100.00	NULL
3	MATERIALIZED	caltrans	NULL	ALL	NULL	NULL	NULL	NULL	103056	2.33	Using where

5. Formal Left Semijoin

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	NULL	ALL	NULL	NULL	NULL	NULL	2402	100.00	Using temporary; Using filesort
2	DERIVED	c	NULL	ALL	PRIMARY	NULL	NULL	NULL	103056	2.33	Using where; Using temporary; Using filesort
3	SUBQUERY	caltrans	NULL	ALL	NULL	NULL	NULL	NULL	103056	2.33	Using where

It seems that the last three queries performed worse than the first two queries, as they have more operations in the evaluation plan. The latter queries generate multiple intermediate tables in order to join or filter the data, which is less efficient than the first two queries. As noted in the solutions to the previous homework, the first two queries took the least amount of time, so they should be the most efficient. This matches the intuition that performing fewer operations in the evaluation plan results in a more efficient query. When comparing the efficiency of the first query to the second, note that the first filters 20.99 percent of 103056 rows, which is about 20 thousand. The second filters 2.33 percent of 103056 rows in one operation and processes all 2402 rows in another without using a where clause. So it seems that the first query matches some duplicate rows before merging the distinct values together. Generating the extra rows leads to a performance penalty that makes its performance comparable to the second query, even though the second query has two operations rather than only one. Note that in this analysis I take the runtime of the query as a measure of efficiency, but I could also measure efficiency through the space required for a query. By generating fewer intermediate tables, the first two queries should take up less space in memory than the remaining queries.

Part 3

Problem 1

Relation R will be scanned once for each iteration of the outer loop, so it will be scanned 100,000 times.

Problem 2

a) Each tuple in L is scanned once in the outer loop. Let $\bar{X}$ be the average size of the result of the index lookup. Then there are $|L|\bar{X}$ output tuples, and so a given tuple in R is scanned on average $\frac{|L|\bar{X}}{|R|}$ times.

b) We are given that our join uses an equality, since the join condition is $t_l.K = t_r.K$. The worst case scenario is that the B$^+$-tree on R is a secondary index, and that K is an attribute that does not form a key for R. Let $\bar{X}$ be the average size of the result of the index lookup. Let b_l be the blocks required to hold the relation L. Let h be the height of the B$^+$-tree on R. We have b_l block transfers in order to read L, h block transfers each loop iteration when finding the first record in the index lookup, and $\bar{X}$ block transfers each loop iteration to find the rest of the index lookup. So in total we have $b_l + |L|(h + \bar{X})$ block transfers.

Problem 3

Consider joining two relations $R \bowtie S$, where b_r is the number of blocks necessary to store relation R and b_s is the number of blocks necessary to store relation S. Generally, block nested-loop join is strictly better than naive nested-loop join, as both have the best case performance of 2 seeks and $b_r + b_s$ block transfers when one relation of the join fits in memory. If neither relation of the join fits in memory, the worst case performance is $b_r + b_r b_s$ block transfers and $2b_r$ disk seeks for block nested-loop join, whereas the worst case performance of naive nested-loop join has $b_r + |R|b_s$ block transfers and $b_r + |R|$ disk seeks. Since $|R| > b_r$, naive nested-loop join will never perform better than block nested-loop join, and we would never want to use it.

When comparing block nested-loop join and indexed nested-loop join, the performance of the indexed nested-loop join depends on what type of index lookup is being performed. Let the number of seeks in an average index lookup on the inner relation be s_l and the number of block transfers in an average index lookup on the inner relation be b_l. Then the worst case performance of indexed nested-loop join is $b_r + |R|b_l$ block transfers and $b_r + |R|s_l$ seeks. The number of seeks in indexed nested-loop join is much higher than for the other nested-loop join methods because it does not necessarily read the inner relation in sequential order. So we would want to use indexed nested-loop join in cases where $|R|b_l$ is much smaller than $b_r b_s$, enough to offset the time to perform additional seeks. If we let the time to perform a block transfer to be t_b and the time for a seek to be t_s, we would want to use an indexed nested-loop join when the following condition holds.

$$|R|(t_b b_l + t_s s_l) < b_r(t_b b_s + t_s)$$

The types of index lookups that would most likely satisfy this condition are a lookup on a primary or secondary B$^+$-tree index with the join condition being an equality on a key, a lookup on a primary B$^+$-tree index with the join condition being an equality on a non-key, and a lookup on a primary B$^+$-tree index with the join condition being a comparison. The first of these types of lookups have a number of seeks and block transfers that scale linearly with the height of the tree, since only one record can be retrieved. So in that case it would be good to use indexed nested-loop join. The remaining two lookups have a number of seeks and block transfers that additionally scale linearly with the number of blocks b necessary to contain the retrieved records. If b is large, it would be better to use block nested-loop join rather than indexed nested-loop join. But it may be better to use an indexed nested-loop join in some cases. This would depend on the data that you are joining. Other types of index lookups such as a comparison with a secondary index may scale badly with the number of records retrieved, so we would prefer to use block nested-loop join if index lookups are too costly.

Problem 4

We can prove the two expressions are equivalent by showing that if there is some tuple α in $E_1 \bowtie_\theta (E_2 - E_3)$, then it must be in $(E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$ and vice versa. Consider the following.

$$\alpha \in E_1 \bowtie_\theta (E_2 - E_3)$$

We know that E_2 and E_3 contain the same set of attributes S since we can apply set difference on them. Then we know that α also contains the set of attributes S. These attributes come from a tuple t in E_2 that is not in E_3, such that t and some other tuple u in E_1 satisfies the join condition θ. Because t is not in E_3, α cannot be in $E_1 \bowtie_\theta E_3$. Additionally, α must be in $E_1 \bowtie_\theta E_2$ because we have already established that there exists some $t \in E_2$

and $u \in E_1$ that satisfy θ and form α. Thus if $\alpha \in E_1 \bowtie_\theta (E_2 - E_3)$, it must be true that $\alpha \in (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.

In the other direction, consider some $\alpha \in (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$. Then $\alpha \in E_1 \bowtie_\theta E_2$ and $\alpha \notin E_1 \bowtie_\theta E_3$. If alpha consists of some tuple t in E_2 joined with a tuple u in E_1, then this means either $t \notin E_3$ or $u \notin E_1$ in order for $\alpha \notin E_1 \bowtie_\theta E_3$. Since we have already established that u is in E_1, this means that $t \notin E_3$. Thus $t \in E_2 - E_3$, and $\alpha \in E_1 \bowtie_\theta (E_2 - E_3)$. Thus these two statements are equivalent.

This rule can be used to improve efficiency of queries by reducing the number of operations necessary to evaluate them. Notice that on the left side of the equation there are only two operations being performed, while there on right side there are three. So converting queries that are in the form on the right into the form on the left will strictly reduce the amount of work necessary to evaluate the query. Instead of doing two joins and one set difference, only one set difference and one join is necessary.

Problem 5

Consider the relations defined by the following attributes.

$$R = (A, B)$$
$$S = (A, B)$$

Let $(1,1) \in R$ and $(1,2) \in S$. Then $R - S$ is $\{(1,1)\}$ and the expression $\Pi_A(R - S)$ is $\{(1)\}$. But we also have $\Pi_A(R) = \{(1)\}$ and $\Pi_A(S) = \{(1)\}$. Thus $\Pi_A(R) - \Pi_A(S) = \{\}$ which is not equal to $\Pi_A(R - S)$.