CSC 370

Activity Worksheet: Accelerating SQL Queries

Mr. Yichun Zhao

Fall 2022

In the previous chapter's worksheets, we determined the answer to several SQL queries defined over Zachary's karate club. The dataset and queries are repeated here for reference.

For each query, you should specify one CREATE INDEX statement that is likely to accelerate the query and explain why you chose that index. The first question is answered already as an example.

Schema

Member(id, name, faction, faction_strength, post_split_club)
Club(id, label)
Faction(id, label)
Relationship(member1, member2, num_contexts)

id	label
1	Mr. Hi
2	John

Table 1: Faction

id	label
1	Mr. Hi's
2	Officers'

Table 2: Club

id	name	faction	fs	psc	id	name	faction	fs	psc
1	Alice	1	strong	1	18	Werner	1	weak	1
2	Bob	1	strong	1	19	Xi	NULL	NULL	2
3	Carol	1	strong	1	20	Yuri	1	weak	1
4	Dave	1	strong	1	21	Zane	2	strong	2
5	Eve	1	strong	1	22	Antonio	1	weak	1
6	Farisha	1	strong	1	23	Babak	2	strong	2
7	Gregoire	1	strong	1	24	Claire	2	weak	2
8	Hamza	1	strong	1	25	Dierdre	2	weak	2
9	Ninad	2	weak	1	26	Einar	2	strong	2
10	Omar	NULL	NULL	2	27	Farouk	2	strong	2
11	Panagiotis	1	strong	1	28	Ghada	2	strong	2
12	Quinn	1	strong	1	29	Hiro	2	strong	2
13	Ravi	1	weak	1	30	Iniko	2	strong	2
14	Saalima	1	weak	1	31	Javier	2	strong	2
15	Tarik	2	strong	2	32	Kumar	2	strong	2
16	Ulysses	2	weak	2	33	Ludmila	2	strong	2
17	Vivienne	NULL	NULL	1	34	Manpreet	2	strong	2

Table 3: Member

m1	m2	nc	m1	m2	nc	m1	m2	nc	m1	m2	nc
1	2	4	1	3	5	1	4	3	1	5	3
1	6	3	1	7	3	1	8	2	1	9	2
1	11	2	1	12	3	1	13	2	1	14	3
1	18	2	1	20	2	1	22	2	1	32	2
2	3	6	2	4	3	2	8	4	2	14	5
2	18	1	2	20	2	2	22	2	2	31	2
3	4	3	3	8	4	3	9	5	3	10	1
3	14	3	3	28	2	3	29	2	3	33	3
4	8	3	4	13	3	4	14	3	5	7	2
5	11	3	6	7	5	6	11	3	6	17	3
7	17	3	9	31	3	9	33	4	9	34	3
10	34	2	14	34	3	15	33	3	15	34	2
16	33	3	16	34	4	19	33	1	19	34	2
20	34	1	21	33	3	21	34	1	23	33	2
24	26	5	24	28	4	24	30	2	24	33	5
24	34	4	25	26	2	25	28	3	25	32	2
26	32	7	27	30	4	27	34	2	28	34	4
29	32	2	29	34	2	30	33	3	30	34	2
31	33	3	31	34	3	32	33	4	32	34	4
33	34	5									

Table 4: Relationship

Questions

1. Interpreting SQL Queries

```
SELECT `name`
FROM `Member`
    JOIN `Faction` ON (`faction` = `Faction`.`id`)
WHERE `faction_strength` = 'weak'
    AND `label` = 'Mr. Hi';
```

Solution:

2. Interpreting SQL Queries

```
SELECT 'M2'.'name'
FROM 'Member' AS 'M1'
    , 'Member' AS 'M2'
WHERE 'M1'.'name' = 'Einar'
  AND 'M1'.'faction' = 'M2'.'faction';
```

Solution:

3. Interpreting SQL Queries

```
SELECT `M1`.`name`, `M2`.`name`
FROM `Member` AS `M1`
    JOIN `Member` AS `M2`
        ON (`M1`.`faction` = `M2`.`faction`)
WHERE `M1`.`post_split_club` <> `M2`.`post_split_club`;
```

Solution:

4. Interpreting SQL Queries

```
SELECT `M1`.`name`, `M2`.`name`
FROM `Member` AS `M1`
    JOIN `Relationship`
        ON (`M1`.`id` = `member1`)
    JOIN `Member` AS `M2`
        ON (`M2`.`id` = `member2`)
WHERE `num_contexts` > 5;
```

Solution:

5. Interpreting SQL Queries

```
SELECT `M2`.`name`
FROM `Member` AS `M1`
    JOIN `Relationship`
        ON (`M1`.`id` = `member1`)
    JOIN `Member` AS `M2`
        ON (`M2`.`id` = `member2`)
WHERE `M1`.`name` LIKE 'f%';
```

Solution:

6. Interpreting SQL Queries

```
SELECT `name`
FROM `Member`
WHERE `faction` = (
    SELECT `id`
    FROM `Faction`
    WHERE `label` = 'Mr. Hi' )
AND `faction_strength` = 'weak';
```

Solution:

7. Interpreting SQL Queries

```
SELECT `name`
FROM `Member`
WHERE `faction` = (
    SELECT `id`
    FROM `Faction`
    WHERE `label` <> 'Ravi' )
AND `faction_strength` = 'weak';
```

Solution:

8. Interpreting SQL Queries

```
SELECT `name`
FROM `Member` AS `M1`
WHERE EXISTS (
    SELECT *
    FROM `Relationship`
        JOIN `Member` AS `M2`
            ON (`id` = `member1`)
    WHERE `M1`.`id` = `member2`
        AND `name` = 'Javier');
```

Solution:

9. Interpreting SQL Queries

```
SELECT `name`
FROM `Member` AS `M1`
WHERE NOT EXISTS (
      SELECT *
      FROM `Relationship`
          JOIN `Member` AS `M2`
               ON (`id` = `member1`)
      WHERE `M1`.`id` = `member2`
           AND `name` = 'Javier');
```

Solution:

10. Interpreting SQL Queries

```
SELECT `label` AS `club`, COUNT(*)
FROM `Member`
    JOIN `Club`
        ON (`post_split_club` = `Club`.`id`)
GROUP BY `label`;
```

Solution:

11. Interpreting SQL Queries

```
SELECT `Club`.`label` AS `club`, COUNT(*)
FROM `Member`
    JOIN `Club`
        ON (`post_split_club` = `Club`.`id`)
    Join `Faction`
        ON (`faction` = `Faction`.`id`)
WHERE (`Club`.`label` LIKE '%Hi%' AND `Faction`.`label` NOT LIKE '
    %Hi%' )
    OR (`Club`.`label` NOT LIKE '%Hi%' AND `Faction`.`label` LIKE '
        %Hi%' )
GROUP BY `Club`.`label`;
```

Solution:

12. Interpreting SQL Queries

```
SELECT `M1`.`id`, `M1`.`name`, `label` AS `faction`, COUNT(*)
FROM `Member` AS `M1`
  JOIN `Relationship`
      ON (`M1`.`id` = `member1`)
  JOIN `Member` AS `M2`
      ON (`M2`.`id` = `member2`)
  JOIN `Faction`
      ON (`M1`.`faction` = `Faction`.`id`)
WHERE `M1`.`faction` <> `M2`.`faction`
GROUP BY `M1`.`id`
HAVING COUNT(*) > 1
ORDER BY COUNT(*) DESC;
```

Solution:

13. Interpreting SQL Queries

```
SELECT `name`, `label` AS `faction`, COUNT(*)
FROM `Member` AS `M1`
    JOIN (
        SELECT `member1` AS `u`, `member2` AS `v`
        FROM `Relationship`
            UNION ALL
        SELECT `member2` AS `u`, `member1` AS `v`
        FROM `Relationship`)
    ON (`u` = `id`)
    JOIN `Faction`
    ON (`faction` = `Faction`.`id`)
GROUP BY `name`
HAVING COUNT(*) >= 5;
```

Solution:

Solutions

Question 1

This query requires no ordering or grouping and has three selection predicates: the join predicate, a selection on ‘faction_strength‘ and a selection on ‘Faction‘.‘label‘. In this case, ‘faction_strength‘ = ’weak’ is a highly selective predicate, so the most effective index is likely to be:

```
CREATE INDEX ‘idx_faction_strength_id_name‘
    ON ‘Member‘(‘faction_strength‘, ‘id‘, ‘name‘);
```

This index contains all the attributes from that table used in the query, ordered by how selective they are and/or how soon they will be used. However, it is highly specialised for this specific query and not useful for any query that does not involve ‘faction_strength‘. A more general choice that might also help a lot would be:

```
ALTER TABLE ‘Member‘
    ADD FOREIGN KEY ‘fk_member_faction‘ ON ‘faction‘ REFERENCES ‘
        Faction‘(‘id‘);
```

This foreign key index will accelerate any joins between these tables, though it is not as effective on this specific query as filtering on the highly selective predicate ‘faction_strength‘ = ’weak’.

Question 2

Here, we have two selection predicates: one on name and one on faction. However, it isn't easy to choose which one to index. If we index on ‘name‘, that helps us with the very selective predicate, ‘name‘ = 'Einar' that only returns one tuple; however, it doesn't help at all to find the tuples in M2 that have the same faction as M1. On the other hand, if we index first on faction and then on name, we can perform the join efficiently, but we don't have the ability to use the index if we filter by name first.

Comparing the two, an index on name only needs to be probed once, together with a full scan to find the matching tuples by faction. An index on faction needs to be probed n times from a scan of Member. Thus, the index on name appears to be better.

```
CREATE INDEX ‘idx_member_name‘ ON ‘Member‘(‘name‘);
```

Question 3

In this query, we have our predicates are on factions matching and post_split_clubs not matching. An index rarely helps for a not-equals operator; so, this one is actually quite easy. We would index on faction (and possibly include post_split_club then name for an additional minor gain at query time but increase in maintenance overhead). In MySQL, every foreign key is indexed.

```
ALTER TABLE `Member`
    ADD CONSTRAINT `fk_member_faction`
        FOREIGN KEY (`faction`) REFERENCES `Faction`(`id`);
```

We could explicitly create the index instead, but we will likely want the foreign key on this table, anyway.

Question 4

Here we have a couple selection predicates: one that the id in member must match member1 or member2 in Relationship, and the other that the num_contexts attribute must be greater than 5. The latter constraint reduces the Relationships table to just 2 tuples instead of 77. Also, ‘id‘ is already indexed in Member because it is a primary key. Thus, the algorithm to answer this query could use an index to find the two relationships with sufficiently many num_contexts using an index that we create and then either do a single scan of the smaller table, Member, or use its primary key index to retrieve the matching names. This will be much more efficient than prioritising the join.

```
CREATE INDEX ‘idx_relationship_num_contexts‘ ON ‘Relationship‘(‘
    num_contexts‘);
```

Question 5

Here we have two predicates again, an equality on the foreign key for the join and a string pattern matching predicate. While it is possible to use indexes effectively with strings, they do not facilitate pattern matching. Moreover, because there is already a primary key index on Member.id, it is difficult to accelerate this query (without using substantially more knowledge that we will gain in the next module of the course). We can already scan Relationship and probe each Member.

Therefore, we are not likely to improve this query with an index. (If we really need to accelerate it, our best choice is a materialised view on the name field).

Question 6

With sub-queries, it becomes much more complex. Here, we could prioritise the sub-query with an index on Faction label or the outer query with an index on faction_strength or faction_strength and faction. It is usually more impact to index the larger relation.

```
CREATE INDEX `idx_member_faction_strength` ON `Member`(`
    faction_strength`, `faction`, `name`);
```

Question 7

In this query, there isn't much hope for accelerating the inner query, because it is based on a not equals operator. We use the same solution as the previous question.

```
CREATE INDEX `idx_member_faction_strength` ON `Member`(`
    faction_strength`, `faction`, `name`);
```

Question 8

With a correlated sub-query, almost without exception, you want to index the correlated attribute, since that is the bottleneck of the query. Here, we clearly index ‘member2‘ as that is what we probe with the attribute from the outer query (M1.id). This one would benefit from including name in the index as well, since it is the only other attribute used.

```
CREATE INDEX ‘idx_relationship_member2_name‘ ON ‘Relationship‘(‘
    member2‘, ‘name‘);
```

Question 9

Although the condition is negated here, we did not optimise the previous query. So, the solution is the same.

```
CREATE INDEX `idx_relationship_member2_name` ON `Relationship`(`
    member2`, `name`);
```

Question 10

We gain more by indexing the larger table, which is Member in this case. (This isn't always true, e.g., when one entire table fits in memory, as we'll discuss in the next module.)

```
CREATE INDEX `idx_member_club` ON `Member`(`post_split_club`);
```

Question 11

We can throw out the string pattern matching as an option. Then we are left again only with the group by key or the join keys. As in the previous example, we benefit most from indexing the largest table, which is again Member.

```
CREATE INDEX `idx_member_club` ON `Member`(`post_split_club`);
```

Question 12

Often it is best to index by a sort key, because an index provides sorted access, but here the sort key is a computed value so we cannot apply this rule of thumb. Indexing by the group by key can also help, especially when filtering by the group size, because all the tuples within the group are clustered together. That doesn't help here, though, because the group by key is a primary key and thus already indexed. Finally, for the join, we can already scan Relationship and probe the primary key index of Member for both FK fields, member1 and member2.

Therefore, we are not likely to see an improvement to this query by adding a new index.

Question 13

This one is awful for accelerating (assuming no optimisation by the query compiler), because any index in the subquery is not available as part of the intermediate result; moreover, the second SELECT scrambles any sorted access that the index might have provided. We are left trying to accelerate the outer query. Here, we already have a primary key index on Member.id that could be probed in the join and the Faction table is tiny.

The only plausible option here would be to use the group by key, but unfortunately the highly selective COUNT(*) cannot be determined without knowing the result a priory of the sub-query.

Thus, this query, too, is unlikely to benefit from the creation of a new index.