CSI 3130 ASSIGNMENT 1

Mikaela Dobie

300164161

Due: Oct 21st, 2023 (Assignment) or Oct 22nd, 2023 (Brightspace)

1. By adding indices to the columns: employees.department\_id and projects.department\_id, the queries could be greatly reduced as both of those columns get iterated through at least twice in the initial query.
   1. Add statement:

CREATE INDEX idx\_e\_dept\_id ON Employees(department\_id);

CREATE INDEX idx\_p\_dept\_id ON Projects(department\_id);

By using the explain clause, the original cost of the query was approximately 581.

After the index statements it became: ~72, using generally populated tables.

* 1. The second optimization you can input is adapting the left join to projects by selecting, aggregating, and grouping the table before joining as the Projects table has no direct connections with the Employees relation.

SELECT

Employees.name,

Departments.name AS department,

select\_projects.project\_count

FROM

Employees

JOIN

Departments ON Employees.department\_id = Departments.department\_id

LEFT JOIN

(Select projects.department\_id, count(projects.project\_id) as project\_count from projects group by projects.department\_id) as select\_projects ON Employees.department\_id = select\_projects.department\_id

ORDER BY

project\_count DESC;

Before indexing, by pushing a select statement it changed the cost from ~581 to ~264 After the indexing, it does still reduce the cost from ~72 to ~30.

Original Relational Algebraic Expression:

ρdepartments.name/department ρcount(projects.project\_id)/project\_count πemployees.name, department, project\_count (τproject\_count (γemployees.name, department ((Employees ⨝employee.department\_id=departments.department\_id Departments) ⟕employee.department\_id=projects.department\_id Projects)))

New Relational Algebraic Expression:

Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
ρdepartments.name/department ρ(σprojects.department\_id, project\_count(Projects)γprojects.department\_id/select\_projects) ρcount(projects.project\_id)/project\_count πemployees.name, department, project\_count (τproject\_count ((Employees ⨝employee.department\_id=departments.department\_id Departments) ⟕employee.department\_id=select\_projects.department\_id (γprojects.department\_id (σprojects.department\_id,project\_count (Projects))))

Ink Drawings
Ink Drawings


1. If both relations each have a sorted set of secondary indices, the first thing to do would be to input them into B+ trees sorted by their secondary indices. To evaluate, one would parse through each item in the tree and compare the values, add the lesser value into the new relation and continue comparing, if a match only one instance is added into the relation. The process is repeated until there are no values left to be added.
2. \*The algorithm from 15.30 has been copied into this assignment and altered to meet the requirements\*

Left outer join:

1. Add a Boolean Value “Flag” to each tuple in relation r set to FALSE.
2. Partition the relation s using hashing function h. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
3. Partition r similarly.
4. For each i:
   1. Load si into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h.
   2. Read the tuples in ri from the disk one by one. For each tuple tr locate each matching tuple ts in si using the in-memory hash index. Flag tr TRUE and output the concatenation of their attributes.
5. For remaining tuples in r with false flags, concatenate each tuple with a null ts and output the result.

Right outer join: \*Note: Same as above, just alterations are done to s instead of r\*

1. Add a Boolean Value “Flag” to each tuple in relation s set to FALSE.
2. Partition the relation s using hashing function h. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
3. Partition r similarly.
4. For each i:
   1. Load si into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h.
   2. Read the tuples in ri from the disk one by one. For each tuple tr locate each matching tuple ts in si using the in-memory hash index. Flag ts TRUE and output the concatenation of their attributes.
5. For remaining tuples in s with false flags, concatenate each tuple with a null tr and output the result.

Full outer join:

1. Add a Boolean Value “Flag” to each tuple in both relations r and s set to FALSE.
2. Partition the relation s using hashing function h. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
3. Partition r similarly.
4. For each i:
   1. Load si into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h.
   2. Read the tuples in ri from the disk one by one. For each tuple tr locate each matching tuple ts in si using the in-memory hash index. Flag tr and ts TRUE and output the concatenation of their attributes.
5. For remaining tuples in r with false flags, concatenate each tuple with a null ts and output the result.
6. For remaining tuples in s with false flags, concatenate each tuple with a null tr and output the result.
7. SELECT A FROM r(A, B, C);

The above is one of the simplest index-only plans that can be made for this question. The point is that any query that only requires access to information that is indexed, does not require the loading of a full table or parsing through it. Thus, the answer to this question is any query that only requires information from the A column as that is the only indexed column in the relation.

1. queries
   1. SELECT \* FROM Account WHERE branch\_name LIKE ‘B%’ and balance IN (SELECT Max(balance) FROM Account GROUP BY branch\_name);
   2. SELECT a2.balance, a2.branch\_name, a2.account\_number FROM account a2 RIGHT JOIN account a1 ON a1.branch\_name = a2.branch\_name and a1.balance<a2.balance WHERE a1.branch\_name LIKE ‘B%’ and a2.balance is not null;
   3. ρaccount/a1 ρaccount/a2 πa2.balance, a2.branch\_name, a2.account\_number(σa2.branch\_name like 'B%' (A1 ⋉a1.balance<a2.balance A2))

Appendices:

Original Student Table:

A computer screen with white text

Description automatically generated

Altered: ALTER TABLE student ADD COLUMN Flag BOOLEAN DEFAULT FALSE;

A black screen with white text

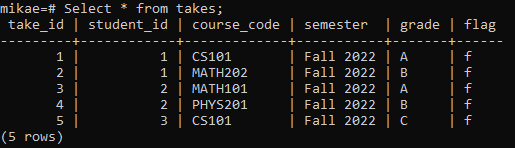
Description automatically generated

Original Takes Table:

A computer screen with white text

Description automatically generated

Altered: ALTER TABLE takes ADD COLUMN Flag BOOLEAN DEFAULT FALSE;



Full outer join of takes and students:

A black screen with white text

Description automatically generated with medium confidence

Employees:

A screen shot of a computer

Description automatically generated

Departments:

A black screen with white text

Description automatically generated

Projects:

A black background with white text

Description automatically generated

Tasks:

A screen shot of a computer

Description automatically generated

Query Plans:

Initial query

---------------------

Sort (cost=283.42..288.37 rows=1980 width=144)

Sort Key: (count(projects.project\_id)) DESC

-> HashAggregate (cost=155.20..175.00 rows=1980 width=144)

Group Key: employees.name, departments.name

-> Hash Right Join (cost=52.05..140.35 rows=1980 width=140)

Hash Cond: (projects.department\_id = employees.department\_id)

-> Seq Scan on projects (cost=0.00..17.20 rows=720 width=8)

-> Hash (cost=45.18..45.18 rows=550 width=140)

-> Hash Join (cost=28.23..45.18 rows=550 width=140)

Hash Cond: (employees.department\_id = departments.department\_id)

-> Seq Scan on employees (cost=0.00..15.50 rows=550 width=72)

-> Hash (cost=18.10..18.10 rows=810 width=72)

-> Seq Scan on departments (cost=0.00..18.10 rows=810 width=72)

(13 rows)

~581

After Index:

----------------

Sort (cost=23.64..23.65 rows=4 width=144)

Sort Key: (count(projects.project\_id)) DESC

-> GroupAggregate (cost=23.52..23.60 rows=4 width=144)

Group Key: employees.name, departments.name

-> Sort (cost=23.52..23.53 rows=4 width=140)

Sort Key: employees.name, departments.name

-> Nested Loop Left Join (cost=1.09..23.48 rows=4 width=140)

Join Filter: (employees.department\_id = projects.department\_id)

-> Hash Join (cost=1.09..22.27 rows=4 width=140)

Hash Cond: (departments.department\_id = employees.department\_id)

-> Seq Scan on departments (cost=0.00..18.10 rows=810 width=72)

-> Hash (cost=1.04..1.04 rows=4 width=72)

-> Seq Scan on employees (cost=0.00..1.04 rows=4 width=72)

-> Materialize (cost=0.00..1.04 rows=3 width=8)

-> Seq Scan on projects (cost=0.00..1.03 rows=3 width=8)

(15 rows)

~72

Added Nested query:

---------------

Sort (cost=98.99..100.37 rows=550 width=144)

Sort Key: select\_projects.project\_count DESC

-> Hash Left Join (cost=55.53..73.96 rows=550 width=144)

Hash Cond: (employees.department\_id = select\_projects.department\_id)

-> Hash Join (cost=28.23..45.18 rows=550 width=140)

Hash Cond: (employees.department\_id = departments.department\_id)

-> Seq Scan on employees (cost=0.00..15.50 rows=550 width=72) -> Hash (cost=18.10..18.10 rows=810 width=72)

-> Seq Scan on departments (cost=0.00..18.10 rows=810 width=72)

-> Hash (cost=24.80..24.80 rows=200 width=12)

-> Subquery Scan on select\_projects (cost=20.80..24.80 rows=200 width=12)

-> HashAggregate (cost=20.80..22.80 rows=200 width=12)

Group Key: projects.department\_id

-> Seq Scan on projects (cost=0.00..17.20 rows=720 width=8)

(14 rows)

~264

Nested query with indices:

------------------

Sort (cost=23.58..23.59 rows=4 width=144)

Sort Key: select\_projects.project\_count DESC

-> Nested Loop Left Join (cost=2.14..23.54 rows=4 width=144)

Join Filter: (employees.department\_id = select\_projects.department\_id)

-> Hash Join (cost=1.09..22.27 rows=4 width=140)

Hash Cond: (departments.department\_id = employees.department\_id)

-> Seq Scan on departments (cost=0.00..18.10 rows=810 width=72)

-> Hash (cost=1.04..1.04 rows=4 width=72)

-> Seq Scan on employees (cost=0.00..1.04 rows=4 width=72)

-> Materialize (cost=1.05..1.12 rows=3 width=12)

-> Subquery Scan on select\_projects (cost=1.05..1.11 rows=3 width=12)

-> HashAggregate (cost=1.05..1.08 rows=3 width=12)

Group Key: projects.department\_id

-> Seq Scan on projects (cost=0.00..1.03 rows=3 width=8)

(14 rows)

~30