Relational algebra

1.

```
-- Yunyu Guo
-- Part 1: For a neighborhood entered by the user, show the popularity of tree species planted in a given year,
-- including the minimum and maximum tree species, associated planting zone factors,
-- filtering the neighborhoods with at least 2 species planted in the given year.
SET @neighborhoodName = ?;
SET @startYear = ?;
SET @endYear = ?;
SET @plantingZoneFactor = ?;

SELECT
   tr.neighborhood,
   COUNT(DISTINCT ts.treeID) AS species_count,
   MIN(ts.commonName) AS min_species,
   MAX(ts.commonName) AS max_species,
   MIN(tp.plantDate) AS firstPlantDate,
   MAX(tp.plantDate) AS lastPlantDate,
   GROUP_CONCAT(DISTINCT tpz.plantingZoneFactor) AS planting_zone_factors -- concatenate values from a group into a single string, separated by commas
From treeRequests tr
JOIN treePlantings tp ON tr.requestID = tp.requestID
JOIN treeSpecies ts ON tp.treePlanted = ts.treeID
LEFT JOIN treeToPlantingZones tpz ON ts.treeID = tpz.treeID
-- WHERE tr.neighborhood = 'Bushrod'
WHERE tr.neighborhood = @neighborhoodName
   AND tr.neighborhood IN (
       SELECT DISTINCT tr2.neighborhood
       FROM treeRequests tr2
       JOIN treePlantings tp2 ON tr2.requestID = tp2.requestID
       -- WHERE YEAR(tp2.plantDate) BETWEEN 2023 AND 2025
       WHERE YEAR(tp2.plantDate) BETWEEN @startYear AND @endYear
       GROUP BY tr2.neighborhood
       HAVING COUNT(DISTINCT tp2.treePlanted) >= 2
   )
 -- AND YEAR(tp.plantDate) BETWEEN 2023  AND 2025
   AND YEAR(tp.plantDate) BETWEEN @startYear AND @endYear
GROUP BY tr.neighborhood

UNION

-- Part 2: filter neighborhoods with a specific planting zone factor entered by the user
SELECT
   tr.neighborhood,
   COUNT(DISTINCT ts.treeID) AS species_count,
   MIN(ts.commonName) AS min_species,
   MAX(ts.commonName) AS max_species,
   MIN(tp.plantDate) AS firstPlantingDate,
```

```
  MAX(tp.plantDate) AS lastPlantingDate,
  GROUP_CONCAT(DISTINCT tpz.plantingZoneFactor) AS planting_zone_factors
FROM treeRequests tr
JOIN treePlantings tp ON tr.requestID = tp.requestID
JOIN treeSpecies ts ON tp.treePlanted = ts.treeID
LEFT JOIN treeToPlantingZones tpz ON ts.treeID = tpz.treeID
-- WHERE tr.neighborhood = 'Bushrod'
WHERE tr.neighborhood = @neighborhoodName
  -- AND tpz.plantingZoneFactor = 'Highly urbanized zones'
 AND tpz.plantingZoneFactor = @plantingZoneFactor
  -- AND YEAR(tp.plantDate) BETWEEN 2023  AND 2025
 AND YEAR(tp.plantDate) BETWEEN @startYear AND @endYear
GROUP BY tr.neighborhood;
```

Subquery: (// Subquery **validneighborhood** for IN clause

ΠTR2.neighborhood(

σTP2.treePlanted >= 2

γTR2.neighborhood, COUNT(DISTINCT treePlanted)

σYEAR(TP2.plantDate) >= @startYear ∧ YEARTP(TP2.plantDate) <= @endYear(

TR2 ⋈ TR2.requestID = TP2.requestID TP2)

Main query part 1:

ΠTR.neighborhood, COUNT(DISTINCT TS.treeID) AS species_count, MIN(TS.commonName) AS min_species, MAX(TS.commonName) AS max_species, MIN(TP.plantDate) AS firstPlantDate, MAX(TP.plantDate) AS lastPlantDate, GROUP_CONCAT(DISTINCT TPZ.plantingZoneFactor) AS planting_zone_factors

γCOUNT(DISTINCT TS.treeID), MIN(TS.commonName), MAX(TS.commonName), MIN(TP.plantDate), MAX(TP.plantDate), GROUP_CONCAT(DISTINCT TPZ.plantingZoneFactor)

σTR.neighborhood = @neighborhoodName ∧ YEAR(TP.plantDate) >= @startYear AND YEAR(TP.plantDate) <= @endYear(

(TR ⋈ TR.requestID=TP.requestID TP ⋈ TP.treePlanted=TS.treeID TS) ⟕ TS.treeID=TPZ.treeID TPZ)

⋉ neighborhood = **validneighborhood**

U

Main query part 2:

ΠTR.neighborhood, COUNT(DISTINCT TS.treeID) AS species_count, MIN(TS.commonName) AS min_species, MAX(TS.commonName) AS max_species, MIN(TP.plantDate) AS firstPlantingDate, MAX(TP.plantDate) AS lastPlantingDate, GROUP_CONCAT(DISTINCT TPZ.plantingZoneFactor) AS planting_zone_factors

γTR.neighborhood

σTR.neighborhood = @neighborhoodName $\wedge$ TPZ.plantingZoneFactor = @plantingZoneFactor

$\wedge$ YEAR(TP.plantDate) >= @startYear $\wedge$ YEAR(TP.plantDate) <= @endYear(

(TR $\bowtie$TR.requestID=TP.requestID TP $\bowtie$ TP.treePlanted=TS.treeID TS) $\bowtie$ TS.treeID=TPZ.treeID TPZ)

2.

```
// get all assigned tree requests and plantings infor for the volunteer
// suppose the volunteer should see the address, phone number, plant date, and tree common name
public void seeTreeRequestAndTreePlantingAndTreeSpecies(int vid) {

  String sql = "SELECT tr.address, tr.phone, tp.plantDate, ts.commonName, vp.requestID " +
        "FROM volunteerPlants vp " +
        "JOIN treePlantings tp ON tp.plantID = vp.plantID " +
        "JOIN treeRequests tr ON tr.requestID = tp.requestID " +
        "JOIN siteVisits sv ON sv.requestNum = tr.requestID " +
        "JOIN recommendedTrees rt ON rt.visitID = sv.siteVisitID " +
        "JOIN treeSpecies ts ON ts.treeID = rt.treeID " +
        "WHERE vp.vid = ?";
```

πtr.address, tr.phone, tp.plantDate, ts.commonName, vp.requestID (

σvp.vid = @volunteerID (

((((vp $\bowtie$ vp.plantID = tp.plantID treePlantings tp)

$\bowtie$ tp.requestID = tp.requestID treeRequests tr)

$\bowtie$tr.requestID = sv.requestNum siteVisits sv)

$\bowtie$ sv.siteVisitID = rt.visitID recommendedTrees rt)

$\bowtie$ rt.treeID = ts.treeID treeSpecies ts

)

)

3.1

```sql
SELECT tr.neighborhood,
    ROUND(AVG(vp.workHour), 2) AS AverageWorkingHour,
    COUNT(DISTINCT tp.requestID) AS treePlantedCount,
    SUM(vp.workloadFeedback = 'overload') AS OverloadCount,
    ROUND(SUM(vp.workloadFeedback = 'overload')/COUNT(*), 2) AS OverloadRate
FROM volunteerPlants vp
  INNER JOIN treePlantings tp ON vp.requestID = tp.requestID
  INNER JOIN treeRequests tr ON tr.requestID = tp.requestID
  INNER JOIN neighborhoods n ON tr.neighborhood = n.name
WHERE n.district = (
     SELECT n2.district
       FROM neighborhoods n2
       INNER JOIN treeRequests tr2 ON tr2.neighborhood = n2.name
       INNER JOIN treePlantings tp2 ON tr2.requestID = tp2.requestID
       INNER JOIN volunteerPlants vp2 ON vp2.requestID = tp2.requestID
     GROUP BY n2.district
     HAVING SUM(vp2.workHour) >= ALL(
       SELECT SUM(vp3.workHour)
         FROM neighborhoods n3
         INNER JOIN treeRequests tr3 ON tr3.neighborhood = n3.name
         INNER JOIN treePlantings tp3 ON tr3.requestID = tp3.requestID
         INNER JOIN volunteerPlants vp3 ON vp3.requestID = tp3.requestID
       GROUP BY n3.district
     )
  )
GROUP BY tr.neighborhood
HAVING COUNT(DISTINCT tp.requestID) > 1;
```

Score of the query: 6 points complex

- Tables joined (1–2:0 points, ≥3:1 point) 1 point
- Non-inner/natural join? (no:0 points, yes:1 point)
- # of subqueries (0:0 points, 1:1 point, >2:2 points) 2 points
- # queries comprising result via union/intersect (0:0 points, ≥1:1 point)
- Aggregate function(s) and grouping rows? (no:0 points, yes:1 point) 1 point
- # WHERE/HAVING conditions not for joins (≤1:0 points, >1:1 point) 1 point
- Non-aggregation functions or expressions in SELECT/WHERE? (no:0 points, yes:1 point)
- Strong motivation/justification for the query in the domain? (no:0 points, yes:1 point) 1 point

subquery_sum_of_workhour_each_district:

$\Pi_{\text{SUM(vp3.workHour)}}\ \gamma_{\text{n3.district,SUM(vp3.workHour)}}$

(

neighborhoods n3

$\bowtie_{\text{tr3.neighborhood=n3.name}}$ treeRequests tr3

$\bowtie_{\text{tr3.requestID=tp3.requestID}}$ treePlantings tp3

$\bowtie_{vp3.requestID=tp3.requestID}$ volunteerPlants vp3
)

subquery_find_max_workhour_district:

$\Pi_{n2.district}$
($\sigma_{SUM(vp2.workHour) \geq ALL(subquery\_sum\_of\_workhour\_each\_district)}$
$\gamma_{n2.district,SUM(vp2.workHour)}$
(neighborhoods n2
$\bowtie_{n2.name=tr2.neighborhood}$ treeRequests tr2
$\bowtie_{tr2.requestID=tp2.requestID}$ treePlantings tp2
$\bowtie_{vp2.requestID=tp2.requestID}$ volunteerPlants vp2
)
)

main_query:

$\Pi_{tr.neighborhood,ROUND(AVG(vp.workHour),2)\text{-> } AverageWorkingHour,}$
COUNT(DISTINCT tp.requestID)-> treePlantedCount,
SUM(vp.workloadFeedback=’overload’) -> OverloadCount,
ROUND(SUM(vp.workloadFeedback=’overload’)/COUNT(*),2)-> OverloadRate
$\sigma_{treePlantedCount>1}$
$\gamma_{tr.neighborhood,ROUND(AVG(vp.workHour),2),COUNT(DISTINCT\ tp.requestID),}$
SUM(vp.workloadFeedback=’overload’),
ROUND(SUM(vp.workloadFeedback=’overload’)/COUNT(*),2
$\sigma_{n.district=(subquery\_find\_max\_workhour\_district)}$
(volunteerPlants vp
$\bowtie_{vp.requestID=tp.requestID}$ treePlantings tp
$\bowtie_{tr.requestID=tp.requestID}$ treeRequests tr
$\bowtie_{tr.neighborhood=n.name}$ neighborhoods n
)

3.2 equivalent RA & SQL query

```
SELECT tr.neighborhood,
    ROUND(AVG(vp.workHour), 2) AS AverageWorkingHour,
    COUNT(DISTINCT tp.requestID) AS treePlantedCount,
    SUM(vp.workloadFeedback = 'overload') AS OverloadCount,
    ROUND(SUM(vp.workloadFeedback = 'overload') * 1.0 / COUNT(*), 2) AS OverloadRate
FROM volunteerPlants vp
  INNER JOIN treePlantings tp ON vp.requestID = tp.requestID
  INNER JOIN treeRequests tr ON tr.requestID = tp.requestID
  INNER JOIN neighborhoods n ON tr.neighborhood = n.name
WHERE n.district IN (
  SELECT district
  FROM (
     SELECT district, SUM(workHour) AS totalWork
     FROM volunteerPlants vp2
        INNER JOIN treePlantings tp2 ON vp2.requestID = tp2.requestID
        INNER JOIN treeRequests tr2 ON tr2.requestID = tp2.requestID
        INNER JOIN neighborhoods n2 ON tr2.neighborhood = n2.name
     GROUP BY district
  ) AS subquery_totalwork_per_district
  WHERE totalWork = (
     SELECT MAX(totalWork)
     FROM (
        SELECT district, SUM(workHour) AS totalWork
        FROM volunteerPlants vp3
           INNER JOIN treePlantings tp3 ON vp3.requestID = tp3.requestID
           INNER JOIN treeRequests tr3 ON tr3.requestID = tp3.requestID
           INNER JOIN neighborhoods n3 ON tr3.neighborhood = n3.name
        GROUP BY district
     ) AS subquery_totalwork_per_district
  )
)
GROUP BY tr.neighborhood
HAVING COUNT(DISTINCT tp.requestID) > 1;
```

subquery_totalwork_per_district:

$\Pi_{\text{district,SUM(workHour) -> totalWork}}$

$\gamma_{\text{district,SUM(workHour)}}$

(volunteerPlants vp3

$\bowtie_{\text{vp3.requestID=tp3.requestID}}$ treePlantings tp3

$\bowtie_{\text{tr3.requestID=tp3.requestID}}$ treeRequests tr3

$\bowtie_{\text{tr3.neighborhood=n3.name}}$ neighborhoods n3

)

)

subquery_max_totalwork_hour:

$\Pi_{\text{MAX(totalWork)}}$ subquery_totalwork_per_district

subquery_max_totalwork_district:
Π_{district} subquery_totalwork_per_district

main_query:

$\Pi_{\text{tr.neighborhood,ROUND(AVG(vp.workHour),2)-> AverageWorkingHour,}}$

COUNT(DISTINCT tp.requestID)-> treePlantedCount,

SUM(vp.workloadFeedback='overload') -> OverloadCount,

ROUND(SUM(vp.workloadFeedback='overload')/COUNT(*),2)-> OverloadRate

$\sigma_{\text{treePlantedCount>1}}$

$\gamma_{\text{tr.neighborhood,ROUND(AVG(vp.workHour),2),COUNT(DISTINCT tp.requestID),}}$

SUM(vp.workloadFeedback='overload'),

ROUND(SUM(vp.workloadFeedback='overload')/COUNT(*),2

$\sigma_{\text{n.district IN (subquery\_max\_totalwork\_district)}}$

(volunteerPlants vp

$\bowtie_{\text{vp.requestID=tp.requestID}}$ treePlantings tp

$\bowtie_{\text{tr.requestID=tp.requestID}}$ treeRequests tr

$\bowtie_{\text{tr.neighborhood=n.name}}$ neighborhoods n

)

Justification:
Version 2 is typically more efficient than Version 1 because:

- It avoids using the ALL clause (which is hard to optimize)
- Uses a MAX() aggregate (which is more efficient)
- Enables better subquery reuse (maybe using a WITH will improve this significantly)

4.1

```
SELECT n.name AS neighborhood, t2.commonName AS mostRecommendedTree
  FROM recommendedTrees rt2
  INNER JOIN treeSpecies t2 ON rt2.treeID = t2.treeID
  INNER JOIN siteVisits sv2 ON rt2.requestID = sv2.requestID
  INNER JOIN treeRequests tr2 ON tr2.requestID = sv2.requestID
  INNER JOIN neighborhoods n ON n.name = tr2.neighborhood
GROUP BY n.name, t2.commonName
HAVING COUNT(*) = (
  SELECT MAX(tree_count)
    FROM (
      SELECT n2.name AS neighborhood, rt3.treeID, COUNT(*) AS tree_count
        FROM recommendedTrees rt3
        INNER JOIN siteVisits sv3 ON rt3.requestID = sv3.requestID
        INNER JOIN treeRequests tr3 ON tr3.requestID = sv3.requestID
        INNER JOIN neighborhoods n2 ON n2.name = tr3.neighborhood
      GROUP BY n2.name, rt3.treeID
    ) AS subquery
  WHERE subquery.neighborhood = n.name
)
ORDER BY n.name;
```

Score of the query: 6 points, complex

- Tables joined (1–2:0 points, ≥3:1 point) 1 point
- Non-inner/natural join? (no:0 points, yes:1 point)
- # of subqueries (0:0 points, 1:1 point, >2:2 points) 2 points
- # queries comprising result via union/intersect (0:0 points, ≥1:1 point)
- Aggregate function(s) and grouping rows? (no:0 points, yes:1 point) 1 point
- # WHERE/HAVING conditions not for joins (≤1:0 points, >1:1 point) 1 point
- Non-aggregation functions or expressions in SELECT/WHERE? (no:0 points, yes:1 point)
- Strong motivation/justification for the query in the domain? (no:0 points, yes:1 point) 1 point

$\Pi_{\text{n.name,t2.commonName}}$

$\sigma_{\text{COUNT(*)=(}}$

$\Pi_{\text{MAX(tree\_count)}}$

$\sigma_{\text{subquery.neighborhood=n.name}}$

(

$\Pi_{\text{n2.name,rt3.treeID,COUNT(*)->tree\_count}}$

$\gamma_{\text{n2.name,rt3.treeID,COUNT(*)}}$

(recommendedTrees rt3

$\bowtie_{\text{sv3.requestID=rt3.requestID}}$ siteVisits sv3

$\bowtie_{\text{tr3.requestID=sv3.requestID}}$ treeRequests tr3

$\bowtie_{\text{tr3.neighborhood=n2.name}}$ neighborhoods n2

)

)-> subquery

)

$\gamma_{n.name,t2.commonName,COUNT(*)}$
(recommendedTrees rt3

$\bowtie_{sv3.requestID=rt3.requestID}$ siteVisits sv3

$\bowtie_{tr3.requestID=sv3.requestID}$ treeRequests tr3

$\bowtie_{tr3.neighborhood=n2.name}$ neighborhoods n2

)

4.2 equivalent RA & query

```sql
SELECT n.name AS neighborhood, t.commonName AS mostRecommendedTree
FROM recommendedTrees rt
INNER JOIN treeSpecies t ON rt.treeID = t.treeID
INNER JOIN siteVisits sv ON rt.requestID = sv.requestID
INNER JOIN treeRequests tr ON tr.requestID = sv.requestID
INNER JOIN neighborhoods n ON n.name = tr.neighborhood
INNER JOIN (
  SELECT n2.name AS neighborhood, rt2.treeID, COUNT(*) AS tree_count
  FROM recommendedTrees rt2
  INNER JOIN siteVisits sv2 ON rt2.requestID = sv2.requestID
  INNER JOIN treeRequests tr2 ON tr2.requestID = sv2.requestID
  INNER JOIN neighborhoods n2 ON n2.name = tr2.neighborhood
  GROUP BY n2.name, rt2.treeID
) AS TreeCounts ON TreeCounts.neighborhood = n.name AND TreeCounts.treeID = rt.treeID
INNER JOIN (
  SELECT SubQuery.neighborhood AS neighborhood, MAX(tree_count) AS max_tree_count
  FROM (
     SELECT n3.name AS neighborhood, rt3.treeID, COUNT(*) AS tree_count
     FROM recommendedTrees rt3
     INNER JOIN siteVisits sv3 ON rt3.requestID = sv3.requestID
     INNER JOIN treeRequests tr3 ON tr3.requestID = sv3.requestID
     INNER JOIN neighborhoods n3 ON n3.name = tr3.neighborhood
     GROUP BY n3.name, rt3.treeID
  ) AS SubQuery
  GROUP BY SubQuery.neighborhood
) AS MaxTreeCounts ON MaxTreeCounts.neighborhood = n.name
WHERE TreeCounts.tree_count = MaxTreeCounts.max_tree_count
GROUP BY n.name, t.commonName
ORDER BY n.name;
```

$\Pi_{n.name,t.commonName}$

$\sigma_{TreeCounts.tree\_count=MaxTreeCounts.max\_tree\_count}$

$\gamma_{n.name,t.commonName}$

(recommendedTrees rt

$\bowtie_{rt.treeID=t.treeID}$ treeSpecies t

$\bowtie_{rt.requestID=sv.requestID}$ siteVisits sv

$\bowtie_{tr.requestID=sv.requestID}$ treeRequests tr

$\bowtie_{tr3.neighborhood=n2.name}$ neighborhoods n

$\bowtie_{TreeCounts.neighborhood=n.name \wedge TreeCounts.treeID=rt.treeID}$

(

$\Pi_{n2.name->neighborhood,rt2.treeID,COUNT(*)->tree\_count}$

$\gamma_{2.name,rt2.treeID,COUNT(*)}$

(recommendedTrees rt2

$\bowtie_{rt2.treeID=sv2.treeID}$ siteVisits sv2
$\bowtie_{tr2.treeID=sv2.treeID}$ treeRequests tr2
$\bowtie_{n2.name=tr2.neighborhood}$ neighborhoods n2
)
)AS TreeCounts
$\bowtie_{MaxTreeCounts.neighborhood=n.name}$
(
$\Pi_{SubQuery.neighborhood->neighborhood,MAX(tree\_count)->max\_tree\_count}$
$\gamma_{SubQuery.neighborhood,MAX(tree\_count)}$
(
$\Pi_{n3.name->neighborhood,rt3.treeID,COUNT(*)->tree\_count}$
(
recommendedTrees rt3
$\bowtie_{rt3.treeID=sv3.treeID}$ siteVisits sv3
$\bowtie_{tr3.treeID=sv3.treeID}$ treeRequests tr3
$\bowtie_{n3.name=tr3.neighborhood}$ neighborhoods n3
)
)AS SubQuery
)AS MaxTreeCounts
)

Justification:
Version 2 is typically more efficient than Version 1 because:

- It avoids correlated subqueries (which are costly and hard to optimize)
- Uses joins with pre-aggregated data (enabling better performance and reuse)
- Allows the optimizer to apply indexes and efficient join strategies
- Performs a fixed number of aggregations, regardless of group count
- Scales better with large datasets due to reduced redundant computation

5.

```sql
-- justification/motivation: we want to track tree species and volunteer number impact in actively planting neighborhood.
-- PS: the current result is empty table as no valid row yet; we temporarily set given year is 2023.
-- Show the number of distinct tree species planted by neighborhood in a given year,
-- only for neighborhoods where total planting events that year exceeded 2,
-- including the number of volunteers involved in each neighborhood’s plantings.
-- time complexity:

SELECT n.name AS neighborhood, COUNT(DISTINCT t.commonName) AS species_count, COUNT(DISTINCT vp.vid) AS total_volunteers
FROM treePlantings tp
  JOIN treeRequests tr ON tp.requestRefNum = tr.referenceNum
  JOIN neighborhoods n ON tr.neighborhood = n.name
  JOIN trees t ON TRIM(LEADING ' ' FROM tp.treePlanted) = t.commonName
  LEFT JOIN volunteerPlants vp ON vp.plantID = tp.plantID -- some tree planting might have no volunteer
WHERE YEAR(tp.plantDate) = 2023 AND tr.neighborhood IN (
  SELECT tr2.neighborhood
  FROM treePlantings tp2
    JOIN treeRequests tr2 ON tp2.requestRefNum = tr2.referenceNum
  WHERE YEAR(tp2.plantDate) = 2023
  GROUP BY tr2.neighborhood
  HAVING COUNT(tp2.plantID) >= 2
)
GROUP BY n.name;
```

1. **Tables joined (1–2: 0 points, ≥3: 1 point):**
    - Main query joins 5 tables. Score: **+1 point**
2. **Non-inner/natural join? (no: 0 points, yes: 1 point):**
    - there is a LEFT JOIN volunteerPlants vp. Score: **+1 point**
3. **# of subqueries (0: 0 points, 1: 1 point, >1: 2 points):**
    - There is one subquery (used in the IN clause). Score: **+1 point**
4. **Aggregate function(s) and grouping rows? (no: 0 points, yes: 1 point):**
    - the main query uses COUNT(DISTINCT ...) aggregate functions and has a GROUP BY n.name. Score: **+1 point**
5. **# WHERE/HAVING conditions not for joins (≤1: 0 points, >1: 1 point):**
    - Main query WHERE: YEAR(tp.plantDate) = 2023 (1), tr.neighborhood IN (...) (2). Score: **+1 point**
6. **Non-aggregation functions or expressions in SELECT/WHERE? (no: 0 points, yes: 1 point):**
    - TRIM(LEADING ' ' FROM tp.treePlanted) is used in a JOIN condition. Score: **+1 point**
7. **Strong motivation/justification for the query in the domain? (no: 0 points, yes: 1 point):**

- The query identifies neighborhoods with a minimum level of planting activity (>=2 plantings) in a specific year (2023) and then calculates the species diversity and volunteer involvement for those same active neighborhoods within that year. This is a plausible analytical question for urban forestry or community programs.Score: **+1 point**

Total Score **7 points**

RA expression:

Subquery IN clause: As **QualifyingNeighborhoods**

QualifyingNeighborhoods = ΠTR.neighborhood σplanting_count >= 2

γTR.neighborhood, COUNT(TP2.plantID) (σYEAR(TP.plantDate) = 2023 (TP $\bowtie$ TP.requestRefNum = TR.referenceNum TR))

Main query:

ΠN.name AS neighborhood, species_count, total_volunteers

γN.name, COUNT(DISTINCT T.commonName) AS species_count, COUNT(DISTINCT VP.vid) AS total_volunteers ((σYEAR(TP.plantDate) = 2023 ((((TP $\bowtie$ TP.requestRefNum = TR.referenceNum TR) $\bowtie$ TR.neighborhood = N.name N) $\bowtie$ TRIM(LEADING ' ' FROM TP.treePlanted) = T.commonName T) ⟕ TP.plantID = VP.plantID VP)) $\ltimes$ TR.neighborhood = **QualifyingNeighborhoods**))

Equivalent RA expressions:

1. Apply filters as early as possible to reduce the size of intermediate relations before joins: Applying the YEAR(tp.plantDate) = 2023 filter (σ) to the TP relation before joining it would reduce the number of rows processed in subsequent joins. Applying the Semijoin (⋉) filter earlier also reduces the data flowing into the later joins.

Π N.name AS neighborhood, species_count, total_volunteers (

γN.name, COUNT(DISTINCT T.commonName) AS species_count, COUNT(DISTINCT VP.vid) AS total_volunteers (

(

(

(

(

(σYEAR(TP.plantDate) = 2023(TP)) //**Filter TP early for main query**

⋈ TP.requestRefNum = TR.referenceNum TR // Join pre-filtered TP with TR

)

⋉ TR.neighborhood = QualifyingNeighborhoods // **Apply IN clause filter Semijoin early**

(

// Start definition of QualifyingNeighborhoods

ΠTR.neighborhood (// Project result of subquery

σplanting_count >= 2 (// Apply Having

γTR.neighborhood, COUNT(TP.plantID) AS planting_count (// Group/Count for subquery

(σYEAR(TP.plantDate) = 2023(TP)) // **Filter TP early for subquery**

⋈ TP.requestRefNum = TR.referenceNum TR // Compute subquery intermediate

)

)

)

// end definition of QualifyingNeighborhoods

)

)

⋈ TR.neighborhood = N.name N // Join with Neighborhoods

)

⋈ TRIM(LEADING ' ' FROM TP.treePlanted) = T.commonName T // Join with Trees

)

⟕ TP.plantID = VP.plantID VP // Left Join with Volunteers

)

) // Grouping and Aggregation applied before final projection

2. Replacing IN with JOIN

   Computes the qualifying neighborhoods first and then uses a standard JOIN instead of a semijoin (⋉) to filter the main query path, because database optimizers often have more sophisticated strategies available for optimizing standard joins compared to semijoins or IN subqueries.

ΠN.name AS neighborhood, species_count, total_volunteers (

γN.name, COUNT(DISTINCT T.commonName) AS species_count, COUNT(DISTINCT VP.vid) AS total_volunteers (// Grouping/Aggregation

(

(

(

(

(

(σYEAR(TP.plantDate) = 2023(TP)) // Filter TP early for main query

⋈ TP.requestRefNum = TR.referenceNum TR // Join pre-filtered TP with TR

)

⋈ TR.neighborhood = QualifyingNeighborhoods // **JOIN with QualifyingNeighborhoods instead of IN**

(

// Start Definition of QualifyingNeighborhoods

ΠTR.neighborhood (

σ planting_count >= 2 (// Apply Having

γTR.neighborhood, COUNT(TP.plantID) AS planting_count (// Group/Count for subquery

(σYEAR(TP.plantDate) = 2023(TP)) // Filter TP early for subquery

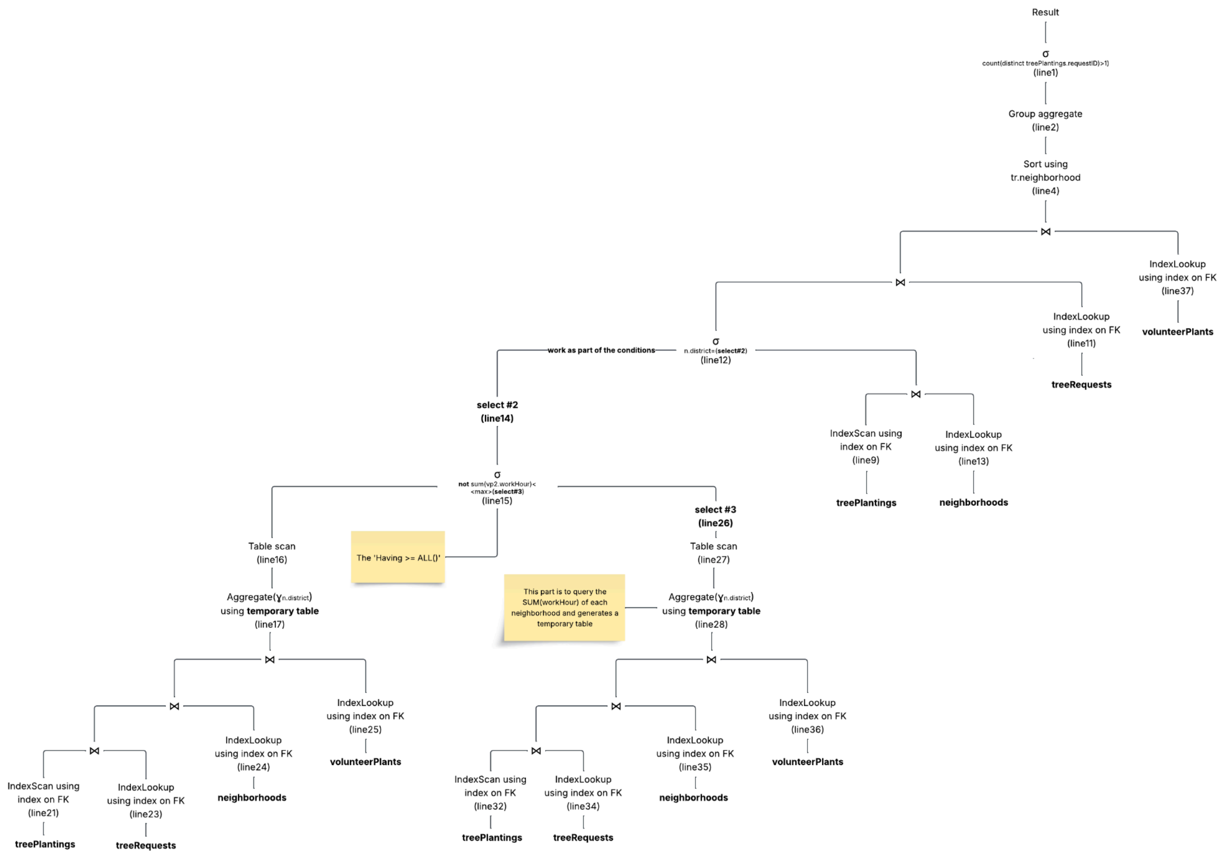
```
                    ⋈ TP.requestRefNum = TR.referenceNum TR  // Compute subquery
intermediate
                   )
                  )
                 )
                 // End Definition of QualifyingNeighborhoods
                )
               )
               ⋈ TR.neighborhood = N.name N                  // Join with Neighborhoods
              )
              ⋈ TRIM(LEADING ' ' FROM TP.treePlanted) = T.commonName T // Join with Trees
             )
             ⟕ TP.plantID = VP.plantID VP                   //Left Join with Volunteers
            )
           )
          )
```

Execution plan and visualization

This is the execution plan that is generated by running `EXPLAIN ANALYZE` on the report query about the workload of volunteers in the district that has the maximum total work hours. For the query please see 3.1 or the first query in reports.sql.

```
-> Filter: (count(distinct treePlantings.requestID) > 1)
    -> Group aggregate: count(distinct treePlantings.requestID), count(0), sum(tmp_field),
       avg(volunteerPlants.workHour), count(distinct treePlantings.requestID), sum(tmp_field)
        -> Sort: tr.neighborhood
            -> Stream results
                -> Nested loop inner join
                    -> Nested loop inner join
                        -> Nested loop inner join
                            -> Covering index scan on tp using FK_planting_aid
                            -> Filter: (tr.neighborhood is not null)
                                -> Single-row index lookup on tr using PRIMARY (requestID = tp.requestID)
                        -> Filter: (n.district = (select #2))
                            -> Single-row index lookup on n using PRIMARY (name = tr.neighborhood)
                            -> Select #2 (subquery in condition; run only once)
                                -> Filter: <not>((`sum(vp2.workHour)` < <max>(select #3)))
                                    -> Table scan on <temporary>
                                        -> Aggregate using temporary table
                                            -> Nested loop inner join
                                                -> Nested loop inner join
                                                    -> Nested loop inner join
                                                        -> Covering index scan on tp2 using FK_planting_aid
                                                        -> Filter: (tr2.neighborhood is not null)
                                                            -> Single-row index lookup on tr2 using PRIMARY (requestID = tp2.requestID)
                                                    -> Single-row index lookup on n2 using PRIMARY (name = tr2.neighborhood)
                                                -> Index lookup on vp2 using PRIMARY (requestID = tp2.requestID)
                                    -> Select #3 (subquery in condition; run only once)
                                        -> Table scan on <temporary>
                                            -> Aggregate using temporary table
                                                -> Nested loop inner join
                                                    -> Nested loop inner join
                                                        -> Nested loop inner join
                                                            -> Covering index scan on tp3 using FK_planting_aid
                                                            -> Filter: (tr3.neighborhood is not null)
                                                                -> Single-row index lookup on tr3 using PRIMARY (requestID = tp3.requestID)
                                                        -> Single-row index lookup on n3 using PRIMARY (name = tr3.neighborhood)
                                                    -> Index lookup on vp3 using PRIMARY (requestID = tp3.requestID)
                    -> Index lookup on vp using PRIMARY (requestID = tp.requestID)
```

And below is the visualization of the execution plan. To review the diagram easily, please use this link to Lucid chart: https://lucid.app/lucidchart/1cdf3ac8-bced-4d66-9733-e5d723fca36c/edit?page=0_0&invitationId=inv_e595e45b-df1f-4694-8fff-eaf8c16fa2be#
We have already sent the invitation to you using the email a.monge@northeastern.edu, if you have not received the invitation, please contact us.



Result
σ
count(distinct treePlantings.requestID)>1)
(line1)
Group aggregate
(line2)
Sort using
tr.neighborhood
(line4)
⋈
IndexLookup
using index on FK
(line37)
volunteerPlants
⋈
IndexLookup
using index on FK
(line11)
treeRequests
σ
n.district=(select#2)
(line12)
work as part of the conditions
⋈
IndexScan using
index on FK
(line9)
treePlantings
IndexLookup
using index on FK
(line13)
neighborhoods
select #2
(line14)
σ
not sum(vp2.workHour)<
<max>(select#3)
(line15)
The 'Having >= ALL()'
Table scan
(line16)
Aggregate(γn.district)
using temporary table
(line17)
⋈
⋈
⋈
IndexScan using
index on FK
(line21)
treePlantings
IndexLookup
using index on FK
(line23)
treeRequests
IndexLookup
using index on FK
(line24)
neighborhoods
IndexLookup
using index on FK
(line25)
volunteerPlants
select #3
(line26)
Table scan
(line27)
This part is to query the SUM(workHour) of each neighborhood and generates a temporary table
Aggregate(γn.district)
using temporary table
(line28)
⋈
⋈
⋈
IndexScan using
index on FK
(line32)
treePlantings
IndexLookup
using index on FK
(line34)
treeRequests
IndexLookup
using index on FK
(line35)
neighborhoods
IndexLookup
using index on FK
(line36)
volunteerPlants

Stored procedure

This stored procedure aims to delete one treeRequest given the requestID.
Analysed steps for the procedure:

1. Verify if this treeRequest exists
2. Find the related siteVisit using the requestID
    a. Find the related recommendedTree data using the requestID
    b. Delete all the related rows in table recommendedTrees
3. Delete the related row in table siteVisits(if exists)
4. Find the related treePlanting using the requestID
    a. Find the related volunteerPlant data using the requestID
    b. Delete all the related rows in table volunteerPlants
5. Delete the related row in table treePlantings(if exists)
6. Delete the treeRequest data in table treeRequests

Since we used ON DELETE CASCADE in the FK constraint of table recommendedTrees and table volunteerPlants, we actually only need to delete the related row in siteVisits and treePlantings.(In table siteVisits and treePlantings the rule is ON DELETE NO ACTION, so we need to delete the data manually)
The code: (you can find this piece of code in the end of file ddl.sql)

```
DROP PROCEDURE IF EXISTS delete_a_treeRequest;
DELIMITER //

CREATE PROCEDURE delete_a_treeRequest(
   IN treeRequestID INT,   -- id of the tree request
   OUT status VARCHAR(500) -- message providing status of request
)
BEGIN
   DECLARE row_count INT DEFAULT 0;
   DECLARE siteVisits_deleted_count INT DEFAULT 0;
   DECLARE treePlantings_deleted_count INT DEFAULT 0;
   DECLARE recommendedTrees_deleted_count INT DEFAULT 0;
   DECLARE volunteerPlants_deleted_count INT DEFAULT 0;

   -- verify if the tree request exists
   SELECT COUNT(requestID) INTO row_count
       FROM treeRequests WHERE requestID = treeRequestID;

   SET status = '';

   IF row_count = 1 THEN -- this tree request exists
       SELECT COUNT(requestID) INTO row_count
           FROM siteVisits WHERE requestID = treeRequestID;

       IF row_count = 1 THEN -- the related site visit exists, delete it
           SET siteVisits_deleted_count = 1;
           -- count the deleted rows in the recommendedTrees
           SELECT COUNT(*) INTO recommendedTrees_deleted_count
```

```
                FROM recommendedTrees WHERE requestID = treeRequestID;
            DELETE FROM siteVisits WHERE requestID = treeRequestID;
        end if;

        SELECT COUNT(requestID) INTO row_count
            FROM treePlantings WHERE requestID = treeRequestID;

        IF row_count = 1 THEN -- the related tree planting event exists, delete
it
            SET treePlantings_deleted_count = 1;
            -- count the deleted rows in the volunteerPlants
            SELECT COUNT(*) INTO volunteerPlants_deleted_count
                FROM volunteerPlants WHERE requestID = treeRequestID;
            DELETE FROM treePlantings WHERE requestID = treeRequestID;
        end if;

        DELETE FROM treeRequests WHERE requestID = treeRequestID;

        SET status = CONCAT(status, 'Deleted the tree request(id): ',
treeRequestID,
                           ', deleted count of rows in siteVisits: ',
siteVisits_deleted_count,
                           ', deleted count of rows in recommendedTrees: ',
recommendedTrees_deleted_count,
                           ', deleted count of rows in treePlantings: ',
treePlantings_deleted_count,
                           ', deleted count of rows in volunteerPlants: ',
volunteerPlants_deleted_count
                    );
    ELSE
        SET status = 'This tree request does not exist!';
    end if;
end //

DELIMITER ;
```

For the Java code that called this stored procedure, please see in the presentation video or task running screenshots in the final report.