RA, TRC, DRC

Created	@April 8, 2025 10:56 AM
Tags	

💾 Schemas:

- `Employee(ename, city, state)`
- `Works(ename, company_name, salary)`
- `Company(company_name, city)`

✅ (a) Relational Algebra Query

Find the names of employees getting salary > 60,000 and working in TCS or WIPRO.

🔍 Interpretation:

We're looking for:

- Employees from `Works` where `salary > 60000`
- And `company_name = 'TCS' OR 'WIPRO'`
- Get their `ename`

✍️ Relational Algebra:

```
text
CopyEdit
π_ename (σ_salary > 60000 ∧ (company_name = 'TCS' ∨ company_name = 'WIPRO') (Works))
```

✅ (b) Tuple Relational Calculus (TRC)

Find employees living in Cuttack and working in companies placed in Bhubaneswar

🔍 Breakdown:

- From `Employee`, find people from `city = 'Cuttack'`
- Join with `Works` and `Company`
- Only if company city is Bhubaneswar

✍️ TRC Query:

```
text
CopyEdit
{ e.ename | Employee(e) ∧ e.city = 'Cuttack' ∧
  ∃ w ∈ Works (e.ename = w.ename ∧
    ∃ c ∈ Company (w.company_name = c.company_name ∧ c.city = 'Bhubane
swar')) }
```

✅ (c) Domain Relational Calculus (DRC)

Find names of employees getting salary more than 60,000

✍️ DRC Query:

```
text
CopyEdit
{ <ename> | ∃ cname, sal (Works(ename, cname, sal) ∧ sal > 60000) }
```

> Here we just project the ename for all tuples in Works where salary > 60000.

🧾 Given Schema:

- `Student(roll_no, name, address)`
- `Marks(roll_no, subject_code, marks)`
- `Subject(subject_code, subject_name)`

✅ (a) Relational Algebra

Find the name of students who have got more than 80 marks in DBMS

Step-by-step:

1. `Subject.subject_name = 'DBMS'`
2. Join it with `Marks` to get those subject codes
3. From `Marks`, select `marks > 80`
4. Join with `Student` to get student names

✍️ Relational Algebra:

```
text
CopyEdit
π_name (
  σ_marks > 80 (
    (Student ⋈ Marks) ⋈ σ_subject_name = 'DBMS' (Subject)
  )
)
```

✅ (b) Tuple Relational Calculus (TRC)

Find students living in Bhubaneswar and scored > 80 in any subject

✍️ TRC Query:

```
text
CopyEdit
{ s.name | Student(s) ∧ s.address = 'Bhubaneswar' ∧
```

```
  ∃ m ∈ Marks (s.roll_no = m.roll_no ∧ m.marks > 80)
}
```

✅ (c) Domain Relational Calculus (DRC)

Find students who have scored > 80 in DBMS

✍️ DRC Query:

```
text
CopyEdit
{ <name> |
  ∃ r, a, sc, sub, m (
    Student(r, name, a) ∧
    Marks(r, sc, m) ∧
    Subject(sc, sub) ∧
    sub = 'DBMS' ∧
    m > 80
  )
}
```

🧠 Declarative vs Procedural

Style	Meaning
Declarative	You **describe what** you want, not how to get it.
Procedural	You **specify how** to get the result step-by-step (like a recipe).

📊 Comparison Table

Feature	Relational Algebra (RA)	Tuple Relational Calculus (TRC)	Domain Relational Calculus (DRC)
Query Style	**Procedural**	**Declarative**	**Declarative**
Unit of operation	Sets / relations	Tuples	Domain values

Focus	How to retrieve data	What data is required	What data is required
Based on	Operations (σ, π, ⋈, etc.)	Predicate logic	Predicate logic
Example Thinking	"Join A and B, then filter"	"There exists a tuple where..."	"There exists a value where..."
Readability (for logic)	Medium	High for logic-based reasoning	High for domain-level queries

📌 Summary:

- **Relational Algebra (RA)** = **Procedural**
- **TRC & DRC** = **Declarative**
- All are part of the **theoretical foundation** of relational databases and query optimization.

Optimizing Tuple Relational Calculus (TRC) queries

This is about **rewriting them for better clarity, logical efficiency**, and often to prepare them for translation into **Relational Algebra or SQL**. Even though TRC is declarative and doesn't specify how to fetch data, we can still make it more efficient logically.

🔧 TRC Optimization Tips

✅ 1. Push Selections Closer to the Source

- If a condition is only on one relation, **don't wrap it in a larger expression.**
- Example:

```
{ t | ∃ e ∈ Employee (t.name = e.name ∧ e.city = 'Cuttack') }
```

➡️ Can be optimized as:

```
{ e.name | Employee(e) ∧ e.city = 'Cuttack' }
```

✅ 2. Avoid Unnecessary Existential Quantifiers

Bad:

```
{ t.name | ∃ e ∈ Employee (t.name = e.name ∧ ∃ w ∈ Works (e.name = w.name ∧ w.salary > 60000)) }
```

Optimized:

```
{ e.name | Employee(e) ∧ ∃ w ∈ Works (e.name = w.name ∧ w.salary > 60000) }
```

> Avoid creating temporary tuple variables (t) when not needed.

✅ 3. Minimize Joins

Combine joins when possible and **project only required attributes**.

Example (original):

```
{ s.name | Student(s) ∧ ∃ m ∈ Marks (s.roll = m.roll ∧ ∃ sub ∈ Subject (m.sub_code = sub.code ∧ sub.name = 'DBMS' ∧ m.marks > 80)) }
```

Optimized:

```
{ s.name | Student(s) ∧ ∃ m ∈ Marks (s.roll = m.roll ∧ m.marks > 80 ∧ ∃ sub ∈ Subject (m.sub_code = sub.code ∧ sub.name = 'DBMS')) }
```

> Rearranged to keep filtering conditions close to their relations.

✅ 4. Use Implicit Projections

If the goal is to **return only one attribute**, declare it clearly:

Instead of:

```
{ t | ∃ s ∈ Student (t.name = s.name ∧ s.address = 'BBSR') }
```

Use:

```
{ s.name | Student(s) ∧ s.address = 'BBSR' }
```

✅ 5. Short-circuit disjunctions or repeated joins

Instead of:

```
{ e.name | Employee(e) ∧ ((∃ w ∈ Works (e.name = w.name ∧ w.company = 'TCS')) ∨ (∃ w ∈ Works (e.name = w.name ∧ w.company = 'WIPRO'))) }
```

Use:

```
{ e.name | Employee(e) ∧ ∃ w ∈ Works (e.name = w.name ∧ (w.company = 'TCS' ∨ w.company = 'WIPRO')) }
```

🚀 Final Tips

- Optimize predicates (`∧`, `∨`, `¬`) logically.
- Minimize nesting and quantifiers.
- Always aim for **readability + minimal joins**.