Joins and semijoins in relational algebra and SQL

Dirk Van Gucht[1]

[1] Indiana University

Joins (Discovering Relationships)

- One of the main purposes of query languages, and more generally search languages, is to discover relationships between objects in the data(base).
- Operations in query languages that can aid in discovering such relationships are called joins

Joins (Discovering Relationships)

- In our running example, the database has three kinds of objects, i.e., students, courses, and teachers, and two kinds of relationships, i.e., Enroll relationships between students and courses, and TaughtBy relationships between courses and teachers.

- Starting from this data, we may then wish to discover each relationship (s, t) between a student s and a teacher t indicating that
  - student s takes some course taught by teacher t, or
  - student s takes all courses taught by teacher t, or
  - student s takes two courses taught by teacher t,
  - etc.

Semijoins (Discovering objects satisfying properties on the basis of relationships)

- Another main purpose of query languages, and more generally search languages, is to discover objects in the database that satisfy certain properties, in particular those that can only be established on the basis of relationships of these objects with other objects in the database.

  For example

  - Find each student s who is enrolled in a class taught by teacher 'Eric'
  - Find each teacher t who teaches no students who take CS courses

- Operations in query languages that can aid in discovering objects with such properties are called semijoins

Joins

- Two types of joins
  - Regular joins:[1] these are joins that compute relationships between objects (o_1, o_2) based on tuple-component comparisons, like $t_1.A\,\theta\,t_2.B$, where o_1 are components of t_1 and o_2 are components of t_2.
  - Set joins: these are joins that compute relations between objects based on comparisons between sets of tuples associated with these objects, e.g, a comparison between the set of courses taken by a student and the set of courses taught by a teacher.
- Analogously, there are also two types of semijoins.

[1] Regular joins are typically just called joins.

Joins in SQL

- Regular joins: SQL supports regular joins with the special operators

JOIN	or, equivalently, INNER JOIN
NATURAL JOIN	special case of JOIN
CROSS JOIN	cartesian product
	special case of NATURAL JOIN

- Set joins: SQL has only limited support for set joins, but they can be simulated using the predicates

  [NOT] EXISTS($\cdots$) and [NOT] IN

  (SELECT COUNT(1) FROM $\ldots$) θ k

Semijoins in SQL

- Regular semijoins: SQL does not have special operations for semijoins, but they can be simulated with the NATURAL JOIN operations, or with the IN set predicate

- Set semijoins: SQL does not have set semijoins, but they can be shown to be special cases of set joins and as such can be simulated

Regular joins in Relational Algebra (Introduction)

- Consider the query "Find the sid and age of each student who takes a course and whose name is Ann."
- This query can be expressed with the RA expression

$$\pi_{\text{Student.sid, age}}(\sigma_{\text{sname}=\text{'Ann'} \wedge \text{Student.sid}=\text{Enroll.sid}}(\text{Student} \times \text{Enroll}))$$

- Or, using the notation $S.sid$ instead of $Student.sid$ and $E.sid$ instead of $Enroll.sid$, more succintly

$$\pi_{\text{S.sid, age}}(\sigma_{\text{sname}=\text{'Ann'} \wedge \text{S.sid}=\text{E.sid}}(\text{Student} \times \text{Enroll}))$$

Regular joins in Relational Algebra (Introduction)

- Consider the query "Find the sid and age of each student who takes a course and whose name is Ann."
- This query can be expressed with the RA expression

$$\pi_{\text{S.sid, age}}(\sigma_{\text{sname='Ann'} \land \text{S.sid=E.sid}}(\text{Student} \times \text{Enroll}))$$

- The subexpression

$$\sigma_{\text{sname='Ann'} \land \text{S.sid=E.sid}}(\text{Student} \times \text{Enroll})$$

  is called a (regular) join and we use the bowtie notation ($\bowtie$) to write it as follows:

$$\text{Student} \bowtie_{\text{sname='Ann'} \land \text{S.sid=E.sid}} \text{Enroll}$$

- With the join operation, we can express the query as follow:

$$\pi_{\text{S.sid, age}}(\text{Student} \bowtie_{\text{sname='Ann'} \land \text{S.sid=E.sid}} \text{Enroll})$$

Regular join in SQL (Introduction)

- "Find the sid and age of each student who takes a course and whose name is Ann."
- In SQL we can formulate this query as follows:

```
SELECT    DISTINCT s.sid, s.age
FROM      Student s JOIN Enroll e
                 ON (s.sname = 'Ann' AND s.sid = t.sid)
```

- As compared to the alternative SQL specification

```
SELECT    DISTINCT s.sid, s.age
FROM      Student s, Enroll e
WHERE     s.sname = 'Ann' AND s.sid = t.sid
```

Regular joins in the Relational Algebra (General case)

- Let $E_1(A_1, \ldots, A_m)$ and $E_2(B_1, \ldots, B_n)$ be RA expressions and let C be a condition involving the attributes $A_1, \ldots A_m$ and $B_1, \ldots, B_n$.
- Then

$$E_1 \bowtie_C E_2$$

  is called the join between E_1 and E_2 on condition C.
- The schema of this join is

$$(E_1.A_1, \ldots, E_1.A_m, E_2.B_1, \ldots, E_2.B_n)$$

- $E_1 \bowtie_C E_2$ is defined to be equivalent with the RA expression

$$\sigma_C(E_1 \times E_2)$$

Regular joins in SQL (General case)

- If Q_{E_1} and Q_{E_2} denote the SQL queries corresponding to E_1 and E_2 then $E_1 \bowtie_C E_2$ corresponds to the SQL query

$$\begin{array}{ll} \text{SELECT} & e_1.A_1, \ldots, e_1.A_m, e_2.B_1, \ldots, e_2.B_n \\ \text{FROM} & (Q_{E_1})\ e_1\ \text{JOIN}\ (Q_{E_2})\ e_2\ \text{ON}\ (\text{C}) \end{array}$$

- If there is no overlap between the schemas of Q_{E_1} and Q_{E_2}, the query can also be formulated as follows:

$$\begin{array}{ll} \text{SELECT} & A_1, \ldots, A_m, B_1, \ldots, B_n \\ \text{FROM} & (Q_{E_1})\ \text{JOIN}\ (Q_{E_2})\ \text{ON}\ (\text{C}) \end{array}$$

More examples

- "Find the sids of students who take at least two courses."
- In RA

$$\pi_{E.sid}(Enroll \bowtie_{E.sid=E_1.sid \wedge E.cno \neq E_1.cno} Enroll_1)$$

- In SQL,

SELECT DISTINCT e.sid
FROM Enroll e JOIN Enroll e_1
ON $(e.sid = e_1.sid \wedge e.cno \neq e_1.cno)$

More examples

- "Find the pids of persons who are not the youngest."
- In RA,

$$\pi_{P.pid}(Person \bowtie_{P.age < P_1.age} Person_1)$$

- In SQL,

SELECT DISTINCT p.pid
FROM Person p JOIN Person p_1 ON ($p.age < p_1.age$)

Special case: cartesian product $\times$

- Let $E_1(A_1, \ldots, A_m)$ and $E_2(B_1, \ldots, B_n)$ be RA expressions
- Then,

$$E_1 \bowtie_{\texttt{true}} E_2 = E_1 \times E_2$$

- In SQL,

$$\begin{array}{ll} \text{SELECT} & A_1, \ldots, A_m, B_1, \ldots, B_n \\ \text{FROM} & (Q_{E_1})\ \text{JOIN}\ (Q_{E_2})\ \text{ON}\ (\text{true}) \end{array}$$

- Alternatively, in SQL, with CROSS JOIN

$$\begin{array}{ll} \text{SELECT} & A_1, \ldots, A_m, B_1, \ldots, B_n \\ \text{FROM} & (Q_{E_1})\ \text{CROSS JOIN}\ (Q_{E_2}) \end{array}$$

Special case: Natural joins (Introduction)

- A natural join between two relations is a series of "equality" joins on the common attributes of the relations, followed by a projection to remove redundant columns.
- Consider the relations Enroll(sid,cno,grade) and TaughtBy(cno,tid); notice that *cno* is a common attribute
- To perform the natural join between these relations, we write $Enroll \bowtie TaughtBy$, where

$$Enroll \bowtie TaughtBy = \pi_{E.sid, E.cno, E.grade, T.tid}(Enroll \bowtie_{E.cno = T.cno} TaughtBy)$$

- In SQL,

```
SELECT   sid, cno, grade, tid
FROM     Enroll NATURAL JOIN TaughtBy
```

Multiple Natural joins

- Recall relations *Enroll*(*sid*, *cno*, *grade*), *Student*(*sid*, *sname*, *age*), and *Course*(*cno*, *cname*, *dept*).
- "Find the sid and sname of each student who takes a CS course."

$$\pi_{sid,sname}(Student \bowtie Enroll \bowtie \sigma_{dept=\text{`CS'}}(Course))$$

- In SQL, this can be formulated as follows

```
WITH      CS_Course AS
            (SELECT * FROM Course WHERE dept = ‘CS’)
SELECT    DISTINCT sid, sname
FROM      Student NATURAL JOIN Enroll
                    NATURAL JOIN CS_Course
```

Natural join (General case)

- Let $E_1(A_1, \ldots, A_m, C_1, \ldots, C_k)$ and $E_2(C_1, \ldots, C_k, B_1, \ldots, B_n)$ be RA expressions
- Then

$$E_1 \bowtie E_2$$

is a natural join expression with schema $(A_1, \ldots, A_m, C_1, \ldots, C_k, B_1, \ldots, B_n)$ and is defined as the RA expression

$$\pi_{A_1,\ldots,A_m,E_1.C_1,\ldots,E_1.C_k,B_1,\ldots,B_n}(E_1 \bowtie_C E_2)$$

where C is the condition

$$E_1.C_1 = E_2.C_1 \wedge \cdots \wedge E_1.C_k = E_2.C_k$$

- Notice that if there are no common C attributes then $E_1 \bowtie E_2 = E_1 \times E_2$; an empty conjunction is interpreted as `true`

Natural joins (General case) in SQL

- If Q_{E_1} and Q_{E_2} denote the SQL queries corresponding to E_1 and E_2 then $E_1 \bowtie E_2$ corresponds to the SQL query

$$
\begin{array}{ll}
\text{SELECT} & A_1, \ldots, A_m, C_1, \ldots C_k, B_1, \ldots, B_n \\
\text{FROM} & (Q_{E_1})\ e_1\ \text{NATURAL JOIN}\ (Q_{E_2})\ e2
\end{array}
$$

SQL as an algebra

- SQL is a declarative language.
- However, with the addition of the JOIN, NATURAL JOIN, and CROSS JOIN, SQL can be seen as an algebra that can faithfully simulate RA
- Indeed,
  - a WHERE clause can simulate a selection operation σ;
  - a SELECT DISTINCT clause can simulate a projection operation π;
  - a CROSS JOIN in the FROM clause can simulate a cartesian product; and
  - SQL has the UNION, INTERSECT, and EXCEPT operator that simulate $\cup$, $\cap$, and $-$

Some observations about the power of the join operator

- Let $E(A_1, \ldots, A_m)$, $E_1(A_1, \ldots, A_m)$, and $E_2(A_1, \ldots A_m)$ be RA expressions with the same schema $(A_1, \ldots, A_m)$
- Then

$$\begin{array}{rcl} E_1 \cap E_2 & = & E_1 \bowtie E_2 \\ \sigma_{A_i \theta \mathbf{a}}(E) & = & \pi_{E.*}(E \bowtie_{E.A_i \theta B} (B : \mathbf{a})) \\ \sigma_{A_i = A_j}(E) & = & \pi_{E.*}(E \bowtie_{E.A_i = E.A_j} E') \end{array}$$

  where E' is a copy of E.
- Conclusion: if we had taken $\bowtie_C$ as a basic operation in RA, then $\times$, $\cap$, and σ become derived operations.
- So all we need are the operations π, $\bowtie_C$, $\cup$, and $-$.
- In SQL, this means that we don't need the WHERE clause!

Semijoins (Introduction)

- "Find the student information for each student who takes a course."

$$\pi_{S.sid,S.sname,S.age}(Student \bowtie Enroll)$$

- In this join expression, we "select" those tuples in the Student relation that join with some tuple in the Enroll relation.
- However, we are not interested in retaining any information (such as cno and grade) from this Enroll tuple.[2]
- This operation is called a semijoin (denoted $\ltimes$) and we simply write

$$Student \ltimes Enroll$$

[2] Observe that $\pi_{S.sid,S.sname,S.age}(Student \bowtie Enroll) \subseteq Student$

Semijoin (Introduction)

Observe that

$$\pi_{S.sid,S.sname,S.age}(Student \bowtie Enroll) = Student \bowtie \pi_{E.sid}(Enroll)$$

- Notice that in the right side of this equation all the unnecessary information from the Enroll relation is projected out. Indeed, only its sid column is relevant in this join.
- We could have therefore also defined the semijoin as follows:

$$Student \ltimes Enroll = Student \bowtie \pi_{E.sid}(Enroll)$$

Semijoins in SQL

- SQL does not have a semijoin operator. However semijoins can be simulated in various ways.
- (1) Using the NATURAL JOIN operator

```
SELECT   sid, sname, age
FROM     Student NATURAL JOIN
             (SELECT DISTINCT sid FROM Enroll) q
```

- (2) Using the IN predicate

```
SELECT   sid, sname, age
FROM     Student
WHERE    sid IN (SELECT DISTINCT sid FROM Enroll)
```

Semijoin (example)

- "Find the student information of students who take a course in the CS department."

$$\pi_{S.sid,S.sname.S.age}(Student \bowtie Enroll \bowtie \sigma_{C.Dept='CS'}(Course))$$

- With semijoins, this becomes

$$Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course))$$

This expression can be evaluated in linear time using hashing!

- Be careful, semijoin is not associative:

$$Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course)) \not\Leftrightarrow (Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course)$$

Semijoin (not associative)

- Be careful, semijoin is not associative:

$$\begin{aligned} &Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course)) \not\Leftrightarrow \\ &\quad (Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course) \end{aligned}$$

- In fact,

$$\begin{aligned} &(Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course)) \Leftrightarrow \\ &\quad Student \ltimes Enroll \end{aligned}$$

if $\sigma_{C.dept='CS'}(Course) \neq \emptyset$.

Semijoin (General case)

- Let $E_1(A_1, \ldots, A_m, C_1, \ldots, C_k)$ and $E_2(C_1, \ldots, C_k, B_1, \ldots, B_n)$ be RA expressions
- Then

$$E_1 \ltimes E_2$$

is the semijoin between E_1 and E_2 (with schema $(A_1, \ldots, A_m, C_1, \ldots, C_k)$) and it is defined as the following RA expression:

$$E_1 \bowtie \pi_{E_2.C_1,\ldots,E_2.C_k}(E_2)$$

- $E_1 \ltimes E_2$ returns each tuples e_1 from E_1 wherefore there exists a tuples e_2 of E_2 such that $e_1.C_1 = e_2.C_1 \wedge \cdots \wedge e_1.C_k = e_2.C_k$.
- So notice that one always has $E_1 \ltimes E_2 \subseteq E_1$.

Semijoins (General case) in SQL

- If Q_{E_1} and Q_{E_2} denote the SQL queries corresponding to E_1 and E_2 then $E_1 \ltimes E_2$ corresponds to the SQL query

$$
\begin{array}{ll}
\text{SELECT} & e_1.* \\
\text{FROM} & (Q_{E_1})\, e_1 \text{ NATURAL JOIN} \\
& \quad (\text{SELECT DISTINCT } C_1, \ldots, C_k \text{ FROM } (Q_{E_2})\, e_2)\; q
\end{array}
$$

- Alternatively, $E_1 \ltimes E_2$ corresponds to the SQL query

$$
\begin{array}{ll}
\text{SELECT} & e_1.* \\
\text{FROM} & (Q_{E_1})\, e_1 \\
\text{WHERE} & (e_1.C_1, \ldots, e_1.C_k) \text{ IN} \\
& \quad (\text{SELECT DISTINCT } e_2.C_1, \ldots, e_2.C_k \text{ FROM } (Q_{E_2})
\end{array}
$$

Semijoin (Time complexity)

- In general, $E_1 \ltimes E_2$ can be implemented (using hasing) in linear time $O(|E_1| + |E_2|)$.
- By contrast, $E_1 \bowtie E_2$ runs in the worst case in $O(|E_1||E_2|)$.
- Intuition by example:
  - Determining if a student takes a CS course can be done fast (linear time)
  - Determining for each student the CS courses taken by that student can be very expensive (quadratic time)
- Lesson: wherever possible, use semijoins

Anti-semijoins

- Consider the query "Find the sid, name, and age of each student who is not enrolled in any course."
- In RA, this query can be formulated as follows

$$Student - Student \ltimes Enroll.$$

- This is correct since $Student \ltimes Enroll$ gives the students who are enrolled in some courses.
- Therefore, if we take the relation Student and subtract from it $Student \ltimes Enroll$, we get indeed the students who are not enrolled in any course.

Anti-semijoins

- "Find the sid, name, and age of each student who is not enrolled in any course". In RA,

$$Student - Student \ltimes Enroll.$$

- There is an operation, called the anti-semijoin, and denoted by $\overline{\ltimes}$, which is especially designed for such expressions.
- Using the anti-semijoin operation $\overline{\ltimes}$, this query can be more succinctly expressed as

$$Student \,\overline{\ltimes}\, Enroll.$$

Anti-semijoins

- In general, given expressions E_1 and E_2, the anti semijoin between E_1 and E_2, i.e., $E_1 \overline{\ltimes} E_2$ is defined as the expression
$$E_1 - (E_1 \ltimes E_2)$$
- In SQL, the anti-semijoin can be supported using the NOT IN set predicate
- Just like the semijoin, the anti-semijoin can be implemented in linear time $O(|E_1| + |E_2|)$ using hashing

Anti-semijoins (General case) in SQL

- If Q_{E_1} and Q_{E_2} denote the SQL queries corresponding to E_1 and E_2, then $E_1 \overline{\ltimes} E_2$ corresponds to the SQL query

$$
\begin{array}{ll}
\text{SELECT} & e_1.* \\
\text{FROM} & (Q_{E_1})e_1 \\
\text{EXCEPT} & \\
\text{SELECT} & e_1.* \\
\text{FROM} & (Q_{E_1})\, e_1 \text{ NATURAL JOIN} \\
 & \quad (\text{SELECT DISTINCT } C_1, ..., C_k \text{ FROM } (Q_{E_2})\, e_2)q
\end{array}
$$

- Alternatively, $E_1 \ltimes E_2$ corresponds to the SQL query

$$
\begin{array}{ll}
\text{SELECT} & e_1.* \\
\text{FROM} & (Q_{E_1})\, e_1 \\
\text{WHERE} & (C_1, \ldots, C_k) \text{ NOT IN} \\
 & \quad (\text{SELECT DISTINCT } C_1, \ldots, C_k \text{ FROM } (Q_{E_2})\, e_2)
\end{array}
$$

Semijoin and Anti-semijoins Special Cases

- Let E_1 and E_2 be expressions with the same schemas
- Then

$$\begin{aligned} E_1 \ltimes E_2 &= E_1 \bowtie E_2 &&= E_1 \cap E_2 \\ E_1 \overline{\ltimes} E_2 &= E_1 - (E_1 \cap E_2) &&= E_1 - E_2 \end{aligned}$$