Relational Calculus

Databases

Pierre Senellart

PSL UNIVERSITÉ PARIS

18 March 2026

Outline

Why study logic?

- Logic is a language to express statements and properties precisely
- It is used throughout mathematics
- It is used throughout computer science:
  - specifications
  - verification
  - artificial intelligence
  - databases
- In databases, logic is used to describe what result we want

Logic is one of the main formal languages of computer science.

A familiar example from mathematics

A standard definition from calculus:

$$\lim_{x \to a} f(x) = \ell$$

means

$$\forall \varepsilon > 0,\ \exists \delta > 0,\ \forall x,\ |x - a| < \delta \Rightarrow |f(x) - \ell| < \varepsilon$$

- $\forall$: "for all"
- $\exists$: "there exists"
- Variables range over some domain
- A formula describes a property that must hold

What first-order logic gives us

The basic ingredients of first-order logic (FO) are:

- variables: $x, y, z \ldots$
- constants: 0, DB, CS...
- predicates / relations: $R(x, y)$, $\mathit{Student}(1, \texttt{Alice}, \texttt{CS})$...
- comparisons: $x = y$, $x < y$, $x \leqslant y$...
- connectives:
  - $\varphi \wedge \psi$ *(and)*
  - $\varphi \vee \psi$ *(or)*
  - $\neg\varphi$ *(not)*
  - $\varphi \Rightarrow \psi$ *(implies)*
- quantifiers:
  - $\exists x\, \varphi$
  - $\forall x\, \varphi$

Logic is interpreted over a structure

A logical formula is not interpreted in the abstract: it is evaluated over some structure.

- In algebra: elements of some algebraic structure (e.g., vectors, matrices), operations of this algebraic structure
- In analysis: reals/complex numbers, addition, multiplication...
- In graph theory: vertices and edges
- In databases: database values; relations used in the logical formulas are the tables of the database (seen as sets of tuples)

A database instance is a finite structure.

Outline

Databases as finite structures

A database schema gives us the relation symbols we will use in our logic (along with comparison symbols such as $=$, $<$, $\leqslant$, etc.).

Example schema:

- `Student(id, name, program)`
- `Course(code, title)`
- `Takes(student, course)`

A database instance gives us the facts:

- which tuples belong to `Student`
- which tuples belong to `Course`
- which tuples belong to `Takes`

Running example

Student

id	name	program
1	Alice	CS
2	Bob	Math
3	Clara	CS
4	David	Physics

Course

code	title
DB	Databases
LOG	Logic
ALG	Algorithms
AI	Artificial Intelligence

Takes

student	course
1	DB
1	LOG
2	LOG
3	ALG
3	DB

Relation facts as atomic formulas

Examples of atomic formulas:

- $\texttt{Student}(1, \text{Alice}, \text{CS})$
- $\texttt{Takes}(1, \text{DB})$
- $\texttt{Course}(\text{DB}, \text{Databases})$

More generally:

$$\texttt{Student}(i, n, p)$$

means that the tuple (i, n, p) belongs to relation `Student`.

We also use comparisons such as:

$$p = \text{CS} \qquad c = \text{DB}$$

A tiny FO toolbox (informal)

- $\varphi \wedge \psi$: both are true
- $\varphi \vee \psi$: at least one is true
- $\neg\varphi$: φ is false
- $\exists x\, \varphi$: there exists a value of x making φ true
- $\forall x\, \varphi$: every value of x makes φ true

Example

$$\exists i \exists p \left(\mathtt{Student}(i, n, p) \wedge p = \mathrm{CS}\right)$$

Free and bound variables

In

$$\exists i \exists p \left(\texttt{Student}(i, n, p) \wedge p = \text{CS} \right)$$

- i and p are bound variables
- n is a free variable

The free variables are what the query returns.

Here, the formula describes the values of n such that there exists a student with name n in the CS program.

Queries in first-order logic

A relational-calculus query has the form

$$Q(x_1, \ldots, x_m) = \varphi$$

where $x_1, \ldots, x_m$ are the free variables of φ.

Meaning:

- input: a database instance D
- output: all tuples $(x_1, \ldots, x_m)$ that make φ true on D

Example

$$Q(n) = \exists i \exists p \left(\texttt{Student}(i, n, p) \wedge p = \text{CS}\right)$$

returns the names of students in the CS program.

Relational calculus

The relational calculus is first-order logic used as a query language over a database schema.

It is:

- a logical formalism
- a declarative formalism
- close in spirit to SQL

A query does not say how to compute the result.
It says which tuples belong to the result.

Relational calculus: syntax

Fix:

- a set $\mathcal{X}$ of variables
- a set $\mathcal{V}$ of values
- a database schema S

Formulas are built as follows:

- if $R \in S$ has arity n, then $R(\alpha_1, \ldots, \alpha_n)$ is a formula
- if $\alpha, \beta \in \mathcal{X} \cup \mathcal{V}$, then $\alpha = \beta$ is a formula (and also $\alpha < \beta$, $\alpha \leqslant \beta$, etc.)
- if φ, ψ are formulas, then

$$\varphi \wedge \psi, \qquad \varphi \vee \psi, \qquad \neg\varphi$$

  are formulas
- if $x \in \mathcal{X}$ and φ is a formula, then

$$\exists x\, \varphi, \qquad \forall x\, \varphi$$

  are formulas

The active domain

Definition

For a database D, the active domain $\mathrm{adom}(D)$ is the set of values that occur in D.

In our running example:

$$\begin{aligned}\mathrm{adom}(D) = \{\, & 1, 2, 3, 4,\\ & \text{Alice, Bob, Clara, David,}\\ & \text{CS, Math, Physics,}\\ & \text{DB, LOG, ALG, AI,}\\ & \text{Databases, Logic, Algorithms, Artificial Intelligence}\,\}\end{aligned}$$

Variables range over actual values occurring in the database.

Relational calculus: semantics

If $Q(x_1, \ldots, x_m) = \varphi$ and D is a database, then

$$Q(D) = \{ (v_1, \ldots, v_m) \in (\text{adom}(D))^m \mid D \models \varphi[x_1/v_1, \ldots, x_m/v_m] \}$$

This means:

- we try all possible values $v_1, \ldots, v_m$
- we substitute them for the free variables (x_i being replaced by v_i)
- we keep the tuples that make the formula true

This just defines the semantics (what the query returns), not the way it is executed in practice!

Satisfaction: the main cases

The notation $D \models \varphi$ means that formula φ is true on database D.

It is defined inductively:

- $D \models R(u_1, \ldots, u_n)$ iff $R(u_1, \ldots, u_n) \in D$
- $D \models u = v$ iff $u = v$
- $D \models \varphi \wedge \psi$ iff $D \models \varphi$ and $D \models \psi$
- $D \models \varphi \vee \psi$ iff $D \models \varphi$ or $D \models \psi$
- $D \models \neg\varphi$ iff $D \not\models \varphi$
- $D \models \exists x\, \varphi$ iff there exists $v \in \mathrm{adom}(D)$ such that $D \models \varphi[x/v]$
- $D \models \forall x\, \varphi$ iff for all $v \in \mathrm{adom}(D)$, $D \models \varphi[x/v]$

De Morgan's laws

Two very useful identities are:

$$\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi \qquad \neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$$

They mean:

- "not (φ and ψ)" means "either not φ, or not ψ"
- "not (φ or ψ)" means "both not φ and not ψ"

Example

$$\neg(p = \text{CS} \vee p = \text{Math}) \equiv (p \neq \text{CS}) \wedge (p \neq \text{Math})$$

These laws, known as De Morgan's laws after the British logician Augustus De Morgan, are constantly used to rewrite logical formulas.

More logical identities

- Two logical identities are used all the time:

$$\forall x\ \varphi \equiv \neg\exists x\ \neg\varphi \qquad\qquad \exists x\ \varphi \equiv \neg\forall x\ \neg\varphi$$

- They mean:
  - "for every x, φ holds" is equivalent to "there does not exist an x such that φ does not hold"
  - "there exists an x such that φ holds" is equivalent to "it is not the case that for every x, φ does not hold"
- We also use implication as a shortcut: $a \Rightarrow b \equiv \neg a \vee b$
- So

$$\texttt{Student}(i, n, p) \Rightarrow \exists c\ \texttt{Takes}(i, c)$$

  is just an abbreviation for

$$\neg\texttt{Student}(i, n, p) \vee \exists c\ \texttt{Takes}(i, c)$$

Outline

A first example

Natural language query:

Return the names of students in the CS program.

Relational calculus:

$$Q(n) = \exists i \exists p \left(\texttt{Student}(i, n, p) \wedge p = \text{CS} \right)$$

The same query in SQL

SQL

```
SELECT DISTINCT s.name
FROM Student s
WHERE s.program = 'CS';
```

- SQL is also declarative
- It describes the desired result
- It does not specify a low-level execution algorithm

Reading the formula aloud

$$Q(n) = \exists i \exists p \left(\texttt{Student}(i, n, p) \wedge p = \text{CS} \right)$$

Read it as:

> *Return every value n such that there exist values i and p for which (i, n, p) is a tuple of relation `Student`, and p is equal to CS.*

A relational-calculus query describes the tuples that should belong to the answer.

A join query

Natural language query:

Return the names of students taking Databases.

Relational calculus:

$$Q(n) = \exists i \exists p \exists c \left(\texttt{Student}(i, n, p) \wedge \texttt{Takes}(i, c) \wedge \texttt{Course}(c, \text{Databases}) \right)$$

The same join query in SQL

SQL

```
SELECT DISTINCT s.name
FROM Student s, Takes t, Course x
WHERE s.id = t.student
  AND t.course = x.code
  AND x.title = 'Databases';
```

- `Student(i,n,p)`
- `Takes(i,c)`
- `Course(c,t)`

Using the same variable twice (i, then c) enforces the equality conditions corresponding to the join.

A disjunctive query

Natural language query:

Return the names of students in CS or Math.

Relational calculus:

$$Q(n) = \exists i \exists p \left(\text{Student}(i, n, p) \wedge (p = \text{CS} \vee p = \text{Math}) \right)$$

SQL

```
SELECT DISTINCT s.name
FROM Student s
WHERE s.program = 'CS' OR s.program = 'Math';
```

A query with negation

Natural language query:

Return the names of students taking no course.

Relational calculus:

$$Q(n) = \exists i \exists p \left(\texttt{Student}(i, n, p) \wedge \neg \exists c \, \texttt{Takes}(i, c) \right)$$

The same query in SQL

SQL

```
SELECT DISTINCT s.name
FROM Student s
WHERE NOT EXISTS (
  SELECT *
  FROM Takes t
  WHERE t.student = s.id
);
```

This is a good example of the close connection between:

- existential quantification in logic
- EXISTS in SQL

Boolean queries

A Boolean query has no free variable.

Example

Is there a student taking Databases?

Relational calculus:

$$\exists i \exists n \exists p \exists c \exists t \left(\texttt{Student}(i, n, p) \wedge \texttt{Takes}(i, c) \wedge \texttt{Course}(c, t) \wedge t = \text{Databases} \right)$$

Universal quantification

We can also write formulas using $\forall$.

Example

Every student takes at least one course.

$$\forall i \forall n \forall p \left(\texttt{Student}(i, n, p) \Rightarrow \exists c \, \texttt{Takes}(i, c) \right)$$

This can also be rewritten without $\Rightarrow$ as:

$$\forall i \forall n \forall p \left(\neg \texttt{Student}(i, n, p) \vee \exists c \, \texttt{Takes}(i, c) \right)$$

Three views of querying

Relational algebra: builds the result with operators

Relational calculus: describes the tuples in the result

SQL: practical declarative language, close in spirit to logic

- Users write SQL
- Internally, the DBMS works with algebraic plans
- The relational calculus is useful to reason about queries, and to abstract away the details of the SQL syntax

Outline

Codd's theorem

Theorem

The relational algebra and the relational calculus have the same expressive power:

- *for every relational algebra query q, there exists a relational calculus query Q such that $q(D) = Q(D)$ for every database D*
- *for every relational calculus query Q, there exists a relational algebra query q such that $Q(D) = q(D)$ for every database D*

Why this theorem matters

- It connects a logical view of querying and an algebraic one
- It explains why a DBMS can:
  - accept a declarative query language
  - translate queries into algebraic expressions
  - optimize and evaluate these expressions
- It is one of the foundational results of relational databases, and one of the reasons of their success

Proof idea

We prove both directions by recursive translation.

Algebra $\rightarrow$ Calculus: translate every algebra operator into logic

Calculus $\rightarrow$ Algebra: for every formula, build an algebra query returning all satisfying assignments

The second direction requires constructing the active domain in algebra.

Proof of algebra $\rightarrow$ calculus

Immediate translation between operators and logical constructs:

- relation $R \longleftrightarrow$ atom $R(\bar{x})$
- selection $\sigma \longleftrightarrow$ conjunction with a condition
- projection $\Pi \longleftrightarrow$ existential quantification
- union $\cup \longleftrightarrow$ disjunction
- difference $\setminus \longleftrightarrow$ conjunction with negation
- join $\bowtie \longleftrightarrow$ conjunction plus equalities

Proof of algebra $\rightarrow$ calculus: all cases

- If $q = R$, define:
$$Q_q(x_1, \ldots, x_n) = R(x_1, \ldots, x_n)$$
- If $q = \sigma_\theta(q')$, define:
$$Q_q(\bar{x}) = Q_{q'}(\bar{x}) \wedge \theta(\bar{x})$$
- If $q = \Pi_{i_1,\ldots,i_k}(q')$, define:
$$Q_q(x_{i_1}, \ldots, x_{i_k}) = \exists \text{ other variables } Q_{q'}(\bar{x})$$
- If $q = q_1 \cup q_2$, define:
$$Q_q(\bar{x}) = Q_{q_1}(\bar{x}) \vee Q_{q_2}(\bar{x})$$
- If $q = q_1 \setminus q_2$, define:
$$Q_q(\bar{x}) = Q_{q_1}(\bar{x}) \wedge \neg Q_{q_2}(\bar{x})$$

Hence every algebra query can be translated into a calculus query.

Proof of calculus $\rightarrow$ algebra: the main difficulty

Suppose $\varphi(x_1, \ldots, x_n)$ is a formula.

We want an algebra query returning:

$$\{(v_1, \ldots, v_n) \mid D \models \varphi[x_1/v_1, \ldots, x_n/v_n]\}$$

This is easy for:

- atomic formulas
- conjunction
- disjunction
- existential quantification

The main issue is negation: we need a relation containing all possible tuples of values.

Building the active domain in relational algebra

Construct a unary relation Dom containing all values appearing in the database.

For our running schema:

$$\begin{aligned}\texttt{Dom} = {} & \Pi_1(\texttt{Student}) \cup \Pi_2(\texttt{Student}) \cup \Pi_3(\texttt{Student}) \\ & \cup \Pi_1(\texttt{Course}) \cup \Pi_2(\texttt{Course}) \\ & \cup \Pi_1(\texttt{Takes}) \cup \Pi_2(\texttt{Takes})\end{aligned}$$

Then $\texttt{Dom}^n$ gives all possible n-tuples of values in the active domain.

Proof of calculus $\rightarrow$ algebra: induction cases

For each formula $\varphi(x_1, \ldots, x_n)$, define an algebra query q_φ returning all satisfying assignments.

All cases:

- if φ is $R(\bar{x})$, use relation R
- if φ is $x = y$, use a selection over $\mathtt{Dom} \times \mathtt{Dom}$
- if $\varphi = \psi \wedge \chi$, join q_ψ and q_χ
- if $\varphi = \psi \vee \chi$, use $q_\psi \cup q_\chi$
- if $\varphi = \neg\psi$, use $\mathtt{Dom}^n \setminus q_\psi$
- if $\varphi = \exists y\, \psi$, project away the column of y
- if $\varphi = \forall y\, \psi$, rewrite as $\neg\exists y\, \neg\psi$

End of the proof

The two recursive translations show that:

- every algebra query can be expressed in the calculus
- every calculus query can be expressed in the algebra

Therefore, relational algebra and relational calculus are equivalent.

This is exactly what makes declarative query languages practical: they can be compiled into algebraic query plans.

Outline

A useful fragment: conjunctive queries

Definition

A conjunctive query (CQ) is a relational-calculus query using only:

- relation atoms and comparisons
- conjunction $\wedge$
- existential quantification $\exists$

and not using:

- disjunction $\vee$
- negation $\neg$
- universal quantification $\forall$

Typical form:

$$Q(\bar{x}) = \exists \bar{y} \, (A_1 \wedge A_2 \wedge \cdots \wedge A_m)$$

Example of a conjunctive query

Our previous join query is a CQ:

$$Q(n) = \exists i \exists p \exists c \left(\texttt{Student}(i, n, p) \wedge \texttt{Takes}(i, c) \wedge \texttt{Course}(c, \text{Databases}) \right)$$

It uses only:

- atoms
- conjunctions
- existential quantifiers

CQs and SQL

Conjunctive queries correspond to the core of SQL (under set semantics, i.e., ignoring duplicates):

- `SELECT`
- `FROM`
- `WHERE`

without:

- subqueries
- `UNION`, `INTERSECT`, `EXCEPT`
- negation such as `NOT EXISTS`
- aggregation

CQs and relational algebra

Conjunctive queries correspond to relational algebra using:

- projection
- selection
- join
- cartesian product

(and renaming, if needed)

but without:

- union
- difference

CQs are simple, but already cover many practical SQL queries.

Outline

First-order logic cannot express everything

First-order logic is expressive, but not all-powerful.

Example schema representing a graph:

$$\texttt{Edge}(x, y)$$

Natural question:

Is there a path from a to b?

This is the reachability problem.

Why reachability is not expressible in FO

Intuition:

- A first-order formula has a fixed finite size
- It can express the existence of a path of length 1, or 2, or 3...
- But it cannot express the existence of a path of arbitrary length
- FO has no recursion or unbounded iteration

To express recursive properties, we need richer query languages.

Outline

Conclusion

- First-order logic is an important language in mathematics and computer science
- The relational calculus is FO interpreted over database instances
- Free variables determine the columns of the answer
- SQL is also declarative and close in spirit to logic
- By Codd's theorem, relational algebra and relational calculus are equivalent
- Conjunctive queries are a central fragment corresponding to select-project-join
- Some natural queries require recursion, beyond FO

Licensing