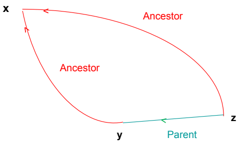
Example:

A Datalog program defining three intentional predicates:

$$\begin{array}{rcl} Person(x) & \leftarrow & Parent(x,y) \\ Person(y) & \leftarrow & Parent(x,y) \\ Grandparent(x,z) & \leftarrow & Parent(x,y), Parent(y,z) \\ Ancestor(x,z) & \leftarrow & Parent(x,z) \\ Ancestor(x,z) & \leftarrow & Ancestor(x,y), Parent(y,z) \end{array}$$

On top of the extensional DB:

Parent	P	C
	juan	pablo
	adam	cain
	adam	abel
	eve	cain
	pablo	luis



Data propagated minimally from right to left, creating (virtual) extensions for intentional predicates

To generate the extension for $Ancestor$, first apply the second last rule, moving all the data from $Parent$ to a partial extension for $Ancestor$, obtaining:

$$Ancestor' = \{(juan, pablo), (adam, cain), (adam, abel), (eve, cain), (pablo, luis)\}$$

Now, use the last, recursive rule, evaluating the RHS, i.e. the query $\Pi_{Anc.1,C}(Ancestor' \bowtie Parent)$ (at this stage a self-join of $Parent$), obtaining a new partial extension for $Ancestor$:

$$Ancestor'' = Ancestor' \cup \{(juan, luis)\}$$

We use the same last rule again, a join of $Ancestor''$ and $Parent$

We evaluate the RHS of the last rule again, but this time nothing new: we have reached a fix-point!

An the minimal model is fully computed

More on Recursion

Example: Descendants of aa?

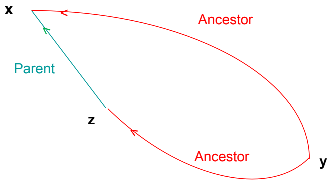
$$Q(x) \leftarrow Ancestor(aa, x) \quad (1)$$

$$Ancestor(x, y) \leftarrow Parent(x, y) \quad (2)$$

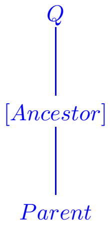
$$Ancestor(x, y) \leftarrow Parent(x, z), Ancestor(z, y) \quad (3)$$

DB $\mathcal{D}$ of Facts: $Parent(a, aa), Parent(a, ab), Parent(aa, aaa),$ $Parent(aa, aab), Parent(aaa, aaaa), Parent(c, ca)$

Like having a selection/projection query (the first rule) on top of a recursively defined view; in its turn defined on top of a RDB



Dependency Graph:



(*Ancestor* is recursive, an iteration)

Computation:

1. Initialize *Ancestor* and Q (query answer predicate) as empty:

   $Ancestor = \emptyset \qquad Q = \emptyset$

2. View *Ancestor* needs to be computed

   First $Ancestor = \emptyset$

   Apply rule (2) once, obtaining by forward propagation:

   $Ancestor = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)\}$

   This is a partial computation of the view

3. Apply (3) with the tuples obtained in the previous step as input for the right-hand side, and propagate to the head

   That is, perform the join $\mathit{Parent} \bowtie$ "$\mathit{Ancestor}$", with "$\mathit{Ancestor}$" only a partial version of the (final) $\mathit{Ancestor}$

   Newly generated tuples for *Ancestor*: $(a, aaa), (a, aab), (aa, aaaa)$

   New state: $\mathit{Ancestor} = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca), (a, aaa), (a, aab), (aa, aaaa)\}$

4. Since new tuples were generated wrt 1., apply rule (3) again, with the partial extension for *Ancestor* as input, from righ to left (forwards) Newly generated tuples for *Ancestor*:

   $\underline{(a, aaa), (a, aab), (aa, aaaa)}, (a, aaaa)$

   The underlined tuples were recomputed!

   New state: $\mathit{Ancestor} = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca), (a, aaa), (a, aab), (aa, aaaa), (a, aaaa)\}$

5. Since new tuples were generated, apply rule (3) once more

   Generated tuples for *Ancestor*:

   $(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)$

6. No new tuples were obtained (redundant recomputation!); same state

   Block for *Ancestor* is completely computed

7. Now compute the extension of Q applying its defining rule (a selection followed by a projection)

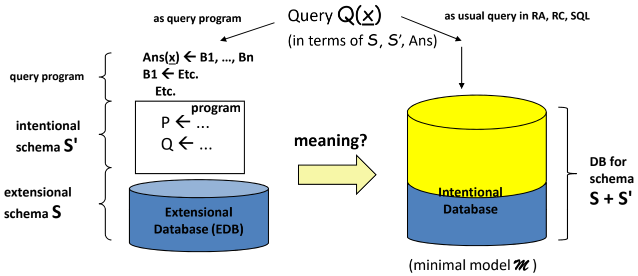
   Generated tuples: $Q = \{aaa, aaaa, aab\}$

Same result is obtained by computing the minimal model for the program consisting of rules (2)-(3) plus $\mathcal{D}$, and posing the top query (1) to the minimal model

The same minimal model can be used for different top queries

On the Minimal Model

- The minimal model can be computed in polynomial time in the size of the extensional DB (EDB) i.e. in data complexity (i.e. varying the EDB, but keeping the program fixed)
- Like creating a new database that extends EDB and querying it as usual



- Alternatively, pose queries to the new virtual database, without materializing first and posing a usual query next

- One can pose the query directly on top of the program that defines the extension, as an extra, top layer

  Propagating upwards data to answer the top query

  But maybe too much data, that is not needed for the top query

- In the example above, the full TC of *Parent* is computed, and moved upwards, to the top query level

  Being only interested in descendants of *aa*, the top selection discards too many computed tuples

- All computations before last step (7.) were done without considering the parameter *aa* in the query

  Many tuples were carried to the upper level and then filtered out: too much useless computation

  This part of the process can be optimized

- There is not only computation of “irrelevant” tuples

  Also recomputation of tuples

- The method applied above for the computation of the minimal model is the “naive method”

  This other part of the process can also be optimized

- With bottom-up evaluation we get all the answers at once

- There are many query optimization methods

Recursion in SQL3

Datalog is built-in in the newer standards of SQL
SQL3 allows to define recursive views, possibly with stratified negation
In a RDBMS using SQL3, we want to answer queries like
Examples:

1. Relation *ParentOf(parent,child)*
   Query: Find all ancestors of Mary
2. "Explosion of Parts"
   Relations: *PartOf(part#,subpart#)*, *Cost(part#,price)*
   Query: What is the total cost of part #123?
   Each part consists of subparts, each subpart of subsubparts, etc.

Example: Ancestors of Mary given the table `ParentOf(parent,child)`?

```
WITH  RECURSIVE Ancestor(anc,desc) AS
      ((SELECT parent AS anc, child AS desc
        FROM ParentOf)
      UNION
      (SELECT Ancestor.anc,
       ParentOf.child AS desc
       FROM Ancestor, ParentOf
       WHERE Ancestor.desc = ParentOf.parent))
SELECT  anc
FROM    Ancestor
WHERE   desc = "Mary";
```

Notice that the definition contains the base case, and the properly recursive case

The union corresponds to the two rules used to define the new predicate

Example: Total cost of part #123 from
PartOf(part#,subpart#) and Cost(part#,price)?

```
WITH      RECURSIVE AllParts AS
          ( (SELECT * FROM PartOf)
          UNION
          (SELECT   A1.part#, A2.part#
          FROM      AllParts A1, AllParts A2
          WHERE     A1.subpart# = A2.part#) )
SELECT    sum(Cost.price)
FROM      AllParts, Cost
WHERE     AllParts.part# = 123
          AND AllParts.subpart# = Cost.part#
```

Several extensions of Datalog

E.g. aggregation functions

R	A	D	N
	a	b	100
	a	c	150
	c	f	30
	c	a	125

$$Ans(x, sum(z)) \leftarrow R(x, y, z)$$

Addition with group-by

Ans	A	N
	a	250
	c	155

Top-Down Query Evaluation

Uses the resolution deductive rule

Program Π:

$$burglary. \quad hearsAlarm(mary). \quad earthquake. \quad hearsAlarm(john).$$

$$\begin{array}{rcll} alarm & \leftarrow & earthquake. & (1) \\ alarm & \leftarrow & burglary. & (2) \\ calls(X) & \leftarrow & alarm, hearsAlarm(X). & (3) \\ call & \leftarrow & calls(X). & (4) \end{array}$$

Positive Datalog

Two ways to evaluate queries:

1. Build the minimal model and query it as a RDB

Bottom-up approach (typical of Datalog)

Minimal Model $\mathcal{M}$ contains:

- Facts: $burglary, \; hearsAlarm(mary), \; earthquake, \; hearsAlarm(john)$
- Derived atoms: $alarm, \; calls(mary), \; calls(john), \; call$

Query: $:\!-\; call?$ *Yes!* (by querying $\mathcal{M}$)

2. Query-dependent methodology based on resolution

   Top-down approach (typical of Prolog)

- Query: *call?* So, we try to prove/deduce atom *call*
- We add it in negated form to the program: $\neg\ call$

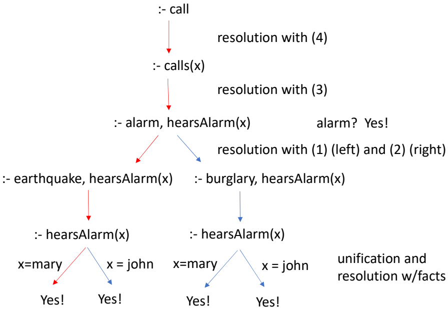
  It will be a proof by contradiction via resolution

  A contradiction is a clause that is always false: the empty clause $\square$ (no literals)
- Good enough: Remember equiconsistency result from Chapter 1

  Negative literal, written in clausal rule form: $\longleftarrow \underbrace{call}_{\text{a goal}}$
- Resolution with (4):

$$\begin{array}{rcl} & \longleftarrow & \not{c}all \\ \not{c}all & \longleftarrow & calls(X) \\ \hline & \longleftarrow & \underbrace{calls(X)}_{\text{new goal}} \end{array}$$

- Etc. until reaching $\square$



- Prolog follows leftmost path in depth (in red) with backtracking
- Search guided by the query (contrary to bottom-up)
- One answer at a time (slower that bottom-up)
- $:-$ the same as $\leftarrow$ (Prolog notation)
- $:- \mathit{call}$ equivalent to $\neg \mathit{call}$
- $:- \mathit{calls}(x)$ equivalent to $\neg \exists x \mathit{call}(x)$
- Above: a successful (resolution-based) refutation tree

- Above *Yes!* means success, i.e. the empty clause $\Box$ was reached
- At the root we have the negation of what we want to prove, in the form: $\leftarrow$ *call*
- The unifications that lead to success are witnesses for the implicit existential variables

  Those values provide the query answers
- Bottom-up vs. Top-down query evaluation?

  Model-based vs. procedural semantics?

  Being true in intended model vs. existence of refutation tree?
- For (positive) Datalog programs Π and conjunctive queries $\mathcal{Q}$ both query evaluation methods are equivalent

  They return exactly the same answers