Logic Databases
(deductive databases)

Incorporates ideas from logic and artificial intelligence.

In addition to storing data, can store rules to infer or deduce new facts.

A declarative language to specify facts and rules.
A variation of Prolog is used (Datalog).

An inference engine is provided to deduce new facts.

Facts are similar to instances of relations in relational databases.

In a RDBMS, the interpretation is suggested by the attribute names; in deductive databases, the interpretation an value depends on its position.

Rules can also be specified. Emphasis in deductive databases has been in the inference of new
facts.

Prolog

A predicate: contains a name and a set of arguments.

If the arguments are constant values (literals) then we are specifying a fact.

If some or all of the arguments are variables then we are specifying a rule.

Constants: denoted by using a lower case letter as the first letter.

```
supervise (jim, john).
supervise (john, jack).
supervise (jack, joe).

superior(X,Y) :- supervise(X,Y).
superior(X,Y) :- supervise(X,Z), superior(Z,Y).
```

Queries can be satisfied by checking for matches in the set of facts, or by applying rules to facts to infer new facts and comparing these to the query goal.

Two resolution techniques are possible:

• Backward chaining (aka Top-down): start with query goal and attempt to match with facts and rules (unification). If solving a goal with a set of subgoals, satisfy in a left-to-right manner.

Pred :- Pred1, Pred2, Pred3, Pred4.

• Forward chaining (aka Bottom-Up): check if facts match query. Then apply each rule to all facts and rules, inferring a set of facts; each inferred fact if compared to the query predicate.

Backward chaining is far more efficient. A large set (potentially infinite) of spurious facts can be generated by the forward chaining techniques.

Prolog systems and deductive databases adopt a backward chaining approach.

The deductive database is queried by specifying a predicate goal.

If the goal contains literals only, then a Boolean value is returned.

If the goal contains variables, then a set of facts are returned that render the query goal true

Safety of Programs

A program (or rule) is said to be safe if it returns a finite set of facts.
To determine if a set of rules is safe or not is undecidable.

Consider the employee schema we met earlier.

We wish to define a rule that returns employees with a large salary

```
large_salary(Y) :- Y > 100000,
                   employee(X),
                   salary (X, Y).
```

Assume we have facts:

employee(jim).
employee(jack).
salary(jim, 30000).
salary(jack, 110000).

Consider:

```
big_salary(Y) :- employee(X),
                 salary (X, Y),
                 Y > 100000.
```

Can also run into problems with recursive rules and ordering of rules and facts.

Consider rules:

...
pred1(X) :- pred(Y).
pred(Y) :- pred1(X).
...

Consider rules:

```
fact (N, X) :-  N1 is N -1,
                fact (N1, X1),
                X is X1 * N.

fact (0, 1).
```

It is clearly important to guarantee safety of programs.

A rule is safe if it generates a finite number of facts.

More formally, a rule is safe if all variables are limited.

A variable X is limited if:

1. it appears in a regular predicate in the rule body

2. it appears in a predicate of form X=c or (c1 <= X and X <= c2) where c, c1 and c2 are constants

3. it appears in a predicate X=Y or Y=X, where Y is a limited variable

Relational Operators

Can easily specify relational operators as Datalog rules.

Allows incorporation of relational views and queries

Example

Assume we have 3 relations each with three arguments
rel_one(A,B,C).
rel_two(D,E,F).
rel_three(G,H,I,J).

Can specify select queries as follows:

```
select_one_A_eq_c(X, Y, Z) :- rel_one(c, Y, Z).

select_one_A_eq_c_and_B_less_5(X, Y, Z) :- rel_one(c, Y, Z),
                                           Y < 5.

select_one_A_eq_c_or_B_less_5(X, Y, Z) :-  rel_one(c, Y, Z).

select_one_A_eq_c_or_B_less_5(X, Y, Z) :-   Y < 5.
```

Can specify project as follows:

project_two_D_F(X,Z) :- rel_three(X,Y,Z).

Set operators:
union_one_two(X,Y,Z) :- rel_one(X,Y,Z).
union_one_two(X,Y,Z) :- rel_two(X,Y,Z).

Note that if rel one and rel two contain matching tuples (facts), we will have duplicates in the result. This isn't strictly correct. Can rewrite as:

union_one_two(X,Y,Z) :- rel_one(X,Y,Z).
union_one_two(X,Y,Z) :- rel_two(X,Y,Z),
not(rel_one(X,Y,Z)).

```
intersect_one_two(X,Y,Z) :- rel_one(X,Y,Z),
rel_two(X,Y,Z).

difference_one_two(X,Y,Z) :- rel_one(X,Y,Z),
                             not(rel_two(X,Y,Z)).


cartesian_one_three(T,U,V,W,X,Y,Z) :-    rel_one(T,U,V),
                                         rel_three(W,X,Y,Z).
```

Hybrid operators (join):

```
join_one_three_C_eq_G(U,V,W,X,Y,Z):-
                                        rel_one(U,V,W),
                                        rel_three(W,X,Y,Z).
```