Other Query Languages

SQL is declarative layer on top of relational algebra. But it is not the only way to query relational data.

Tuple Relational Calculus

Basically uses first-order logic to construct relations.

A query, as in relational algebra, consists in creating a relation that contains the data you are interested in.

How do we create relations using tuple relational calculus?

$$\{t \mid P(t)\}$$

Recall that a relation is a set of tuples. The above expression tells you that the relation is the set of all tuples t that satisfy property P. Property $P(t)$ is generally taken to be a closed formula of first-order logic with t as a free variable, with individuals consisting of values and tuples over those values, and a first-order vocabulary consisting of predicates $\in$, $=$, and terms $t[A]$ representing attribute A's value of tuple t.

Consider the following relations, which we've used as examples in the past few lectures:

Account:

account	type	balance
991	CH	1000
992	SA	500
993	CH	50
994	CH	100
995	CD	10000

Client:

ssn	name
1111	R
2222	T
3333	A

HasAccount:

account	ssn
991	1111
992	1111
993	2222
994	1111
994	3333
995	1111

Query for all accounts with a balance of at least 1000:

$$\{t \mid t \in \mathsf{Account} \wedge t[\text{account}] \geq 1000\}$$

yielding:

991	CH	1000
995	CD	10000

Query for only the account number of the accounts with a balance of at least 1000:

$$\{t \mid \exists s \in \mathsf{Account}\ (t[\text{account}] = s[\text{account}] \wedge t[\text{account}] \geq 1000)\}$$

yielding:

991
995

(Intuitively, the schema of the resulting relation is derived from the attributes for tuple t showing up in the property, where a term of the form $t \in R$ implies that all the attributes in the schema of R show up in the property for t.)

Query for ssn of all clients with a savings account:

$$\{t \mid \exists s \in \mathsf{HasAccount}\ (t[\text{ssn}] = s[\text{ssn}] \wedge \exists u \in \mathsf{Account}\ (u[\text{account}] = s[\text{account}] \wedge u[\text{type}] = \text{SA}))\}$$

yielding:

1111

Query for name of all clients with a savings account:

$$\{t \mid \exists s \in \mathsf{Client}\ (t[\text{name}] = s[\text{name}] \wedge \exists u \in \mathsf{HasAccount}\ (u[\text{ssn}] = s[\text{ssn}] \wedge \exists v \in \mathsf{Account}\ (v[\text{account}] = u[\text{account}] \wedge v[\text{type}] = \text{SA})))\}$$

Not all properties define finite relations. For instance, the query

$$\{t \mid \neg(t \in \mathsf{Clients})\}$$

is the relation where t is a tuple that's not in the Client relation. That includes an infinite number of tuples, including tuples with an arbitrary number of elements.

In order to properly define a finite relation, the property $P(t)$ is required to be *safe*. Safety is a technical condition, but in short, you can think of it that way. Given a property P, define the *domain* of P to be the set of all values referenced by P (the values of all the attributes of all the tuples appearing in relations referred to in P) as well as all the constants appearing in P. Convince yourself that if every relation has only a finite number of tuples, then the domain of P must be a finite set. We define a property P to be safe if the tuples satisfying property P all take their values in the domain of P.

Datalog

Comes from deductive databases. Alternative to SQL that lets you write recursive queries. Especially useful for recursive (hierarchical data).

Sample employee hierarchy:

Employee:

name	manager	daysWorked
b	a	100
x	b	150
y	b	200
c	a	250
d	c	300
u	d	350
v	d	400

Note that datalog relies on accessing attributes of a query by position, and not by name.

A datalog query is expressed as asking for all tuples of a relation that match a certain template. For example, asking for all the tuples in Employee where b is the manager:

```
Employee(X,b,D).
```

Capital letters are placeholder variables. This asks for all tuples where b is in the second (manager) position. The result is usually expressed as:

```
X = x , D = 150
X = y , D = 200
```

To write more elaborate queries, datalog lets us express intermediate relations that you can query. These intermediate relations are expressed using *deductive rules*. For instance, you can express a relation capturing an employee's manager's manager's as:

```
EmployeeManagerManager(X,Y) :- Employee(X,Z,D), Employee(Z,Y,E)
```

Read this from right to left: if X is an employee with manager Z, and Z is an employee with manager Y, then Y is X's manager's manager. (Another reading is from left to right: Y is X's manager's manager when X is an employee with manager Z (for some Z) and Z is an employee with manager Y. Pick whichever works best for you.)

You can then query for a manager's direct reports' direct reports with:

```
EmployeeManagerManager(X,c)
```

which should return:

```
X = u
X = v
```

We require rules to be *range restricted*: every variable appearing in the head of a rule (before the `:-`) must also appear in the body of the rule (after the `:-`).

Rules can be recursive, which lets you express a relation capturing whether an employee is a report of another, meaning that there is a chain of manager linking both employees:

```
Reports(X,Y) :- Employee(X,Y,D)
Reports(X,Y) :- Employee(X,Z,D), Reports(Z,Y)
```

You can read this from left to right as X reports to Y if Y is X's manager, *or* when Z is X's manager (for some Z) and Z reports to Y.

If you were to explicitly construct relation Reports, it would look like:

b	a
x	a
y	a
c	a
d	a
u	a
v	a
x	b
y	b
d	c
u	c
v	c
u	d
v	d

Another example: determining the different in levels between two employees:

```
Difference(X,Y,1) :- Employee(X,Y,D)
Difference(X,Y,N) :- Employee(X,Z,D), Difference(Z,Y,M), N = M+1
```

Querying for `Difference(u,b,D)` should yield the difference in levels between u and b, which is `D = 2`.

Example: two employees are teammates if they have the same manager:

```
Teammates(X,Y) :- Employee(X,Z), Employee(Y,Z)
```

How do we think about the relations constructed by datalog rules? One way to think about it is in terms of something like the tuple relational calculus. If you consider the rules of relation Reports above:

```
Reports(X,Y) :- Employee(X,Y,D)
Reports(X,Y) :- Employee(X,Z,D), Reports(Z,Y)
```

and you reflect about the "meaning" that we gave them, you can think of them as defining a relation Reports defined by the following property:

$$\begin{aligned}\mathsf{Reports} = \{t \mid {} & \exists s \in \mathsf{Employee}\ (t[0] = s[0] \wedge t[1] = s[1]) \\ & \vee \exists s \in \mathsf{Employee}\ \exists u \in \mathsf{Reports}\ (t[0] = s[0] \wedge t[1] = u[1] \wedge s[1] = t[0])\}\end{aligned} \tag{1}$$

Note that this is not a proper definition — you cannot define a set in terms of itself. The above is an equation, and a solution to the equation is a set Reports that satisfies the equation.

How do you solve such an equation? It's usually tricky business. But in this case, the properties corresponding to the datalog rules we've seen all have a monotonicity property: adding more tuples to base relations only lead to more tuples added to any potential solution to the equation, and never fewer. This means that we can use a simple least-fixed-point computation to get a solution:

- let $S = \varnothing$ – this is our target solution
- compute S' to be the result obtained from equation 1 replacing Reports in the property by S, and take the new S to be the old S unioned with S'.
- repeat until nothing gets added to S, and take Reports to be the set of tuples S.

Datalog can express all relational algebra operators. Assume R has schema (a, b), S has schema (a, b) and T has schema (c, d):

- $X = \pi_a(R)$ can be expressed by `X(A) = R(A,B)`
- $X = \sigma_{a=c}(R)$ can be expressed by `X(A,B) = R(A,B), A=c`
- $X = R \cup S$ can be expressed by `X(A,B) = R(A,B)` and `X(A,B) = S(A,B)`
- $X = R \times T$ can be expressed by `X(A,B,C,D) = R(A,B), T(C,D)`

Negation is a feature that is useful in a query language, but is also tricky. For instance, you could write a query that says that there is no reporting relation between two employees:

```
NotReports(X,Y) :- Employee(X,Z1,D1), Employee(Y,Z2,D2), not Reports(X,Y).
```

One restriction on negation is that a variable can only appear in a negated atom (a component of the body of a rule) when it also appears in a non-negated atom. Such a rule is said to be safe (and this kind of safety is similar to that of the tuple relational calculus).

Another restriction on negation is that the rules must be *stratified*. This is to ensure that the process we described earlier to find a solution to equations obtained from datalog rules still applies. Negation destroys monotonicity, and stratification basically ensure enough monotonicity for the least-fixed-point computation to proceed.

Intuitively, statification means partitioning the relations defined by a set of rules into different strata, where the rules defining relations in one strata can only use negated atoms involving rules in a lower stratum.

More formally, say that a relation R depends positively on a relation S if every rule for R that refers to S only uses S without negating it. Say that a relation R depends negatively on a relation S if some rule for R refers to S under a negation.

A stratification for a set of relations defined by rules is a partition of the relations into strata $1, \ldots, N$ such that whenever R depends positively on S, then the stratum of R is greater than or equal to the stratum of S, and whenever R depends negatively on S, then the stratum of R is strictly greater than the stratum of S.

This means that we can determine the tuples in all the relations in statum 1 first. Then use that to determine the tuples in the relations in stratum 2, and so on. If we ever need to determine that a tuple is *not* in a given relation S when defining relation R, then since S is required to live in a previous stratum, we already know the content of S, and therefore can safely determine whether the tuple is not in S without worrying that the construction will later add the tuple to S and contradict our results.

Stratification is also useful when considering aggregated Datalog, which is an extension of Datalog that allows aggregation. It can be written as follows:

```
AvgDaysWorkedDirectReports(X,AVG(<D>)) :- Employee(Y,X,D)
```

This is roughly equivalent to the SQL query

```
SELECT manager, AVG(daysWorked) AS avgDaysWorkd
FROM Employee
GROUP BY manager
```

(minus the naming of the attributes in the resulting relation).

Again, this can only be made sense of if the relation being aggregated over is fully known when computing the aggregation. Therefore, the rules must again be stratified, with a variant definition of stratification with the additional clause: whenever R depends aggregatively on S, then the stratum of R is strictly greater than the stratum of S. A relation R depends aggregatively on relation S if some rule defining R with an aggregation operator over some variable X refers to S in the body of the rule using variable X.