4 Query Flocks

Goal: apply a-priori trick and other association-rule tricks to a more general class of complex queries.

4.1 Query Flock Notation

A *query flock* is a generate-and-test system consisting of:

1. A query with parameters; we write the query in Datalog to simplify certain optimizations later.
2. A filter condition that says when the values of the parameters yields a query result that we accept.

- Note that the query flock is really a single query about its parameters; the parametrized-query component is *not* the real query.

Example 4.1: Frequent item pairs in a relation $Baskets(BID, item)$ can be written as the query flock:

```
Answer(b) <- Baskets(b,$1) AND Baskets(b,$2)

COUNT(Answer) >= s
```

If we replace parameters \$1 and \$2 by values, e.g., "diapers" and "beer," respectively, then the query is asking for the set of basket ID's such that the basket contains both diapers and beer. The condition on the answer says that there must be at least s such baskets, where s is the support threshold. Thus, this query flock asks the usual question about the parameters \$1 and \$2: "which pairs of items appear in at least s baskets?" □

Example 4.2: Here is a less usual example. It supposes relations:

1. $Cust(name, attr, value)$. Tuple (n, a, v) means the customer with name n has value v for attribute a. For instance, $(Sue, age, 45)$ means that Sue is of age 45.
2. $Buys(name, prod)$ tells what products each customer buys.
3. $Type(prod, type)$ tells the type of each product, e.g., product "Coke" is of type "soft drink."

Here is the query flock that asks for values of some attribute that occur at least s times among buyers of a certain type of product:

```
Answer(n) <- Cust(n,$a,$v) AND Buys(n,p) AND Type(p,$t)

COUNT(Answer) >= s
```

□

4.2 Execution Strategies

The analog of a-priori is the observation that if we delete one or more subgoals from a Datalog query, the size of the set of answers can only increase. Our hope is that by computing some temporary relations using a subset of the subgoals, we can filter the sets of values for one or more parameters, using computations that are much less expensive than computing the entire query about the full set of parameters.

We can describe the intermediate steps, as well as the final computation of the parameter-values that pass the test by a sequence of steps of the form

```
<Relation> := FILTER(<parameters>, <query>, <condition>)
```

- The query is the flock query, with zero or more subgoals eliminated. A requirement is that this query be *safe*; i.e., every variable appearing in the head appears in a nonnegated subgoal involving a relation (i.e., not a subgoal involving an arithmetic comparison like $a < b$).

- The parameters are those appearing in the query.
- The condition is the same as the condition of the flock itself.

Example 4.3: The flock of Example 4.1 might be solved by using the first subgoal to filter \$1 and the second subgoal to filter \$2.

```
OK1($1) := FILTER({$1}, Answer(b) <- Baskets(b,$1), COUNT(Answer) >= s)
OK2($2) := FILTER({$2}, Answer(b) <- Baskets(b,$2), COUNT(Answer) >= s)
OK($1,$2) := FILTER({$1,$2}, Answer(b) <- Baskets(b,$1) AND
                             Baskets(b,$2) AND OK1($1) AND OK2($2),
                     COUNT(Answer) >= s)
```

- Of course a clever flocks compiler recognizes that these two filtering steps are really the same and only computes one of OK1 and OK2.
- The reason a-priori often saves a lot of time is because the join of four relations at the last step [computation of $OK(\$1,\$2)$] can be carried out in an order that reduces the size of intermediate relations, when compared with just joining *Baskets* with itself, as suggested by the ordering of Fig. 6.

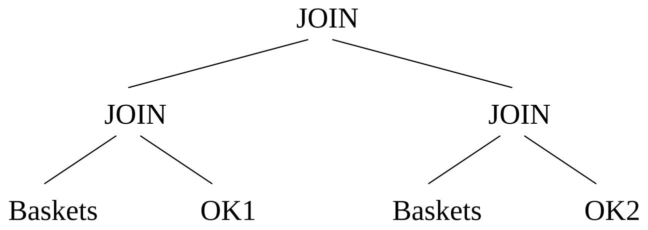

Figure 6: Preferred order for join in market-basket flock

- Notice that the ordering in Fig. 6 is not a left-deep ordering, which suggests that the typical commercial DBMS would *not* find this order, and a query-flocks compiler needs to feed simpler queries to the DBMS so the right order of join is used by the DBMS.

□

Example 4.4: Now let us consider how we might use filter steps to improve the running time of the final join in Example 4.2. Using just the *Cust* subgoal is a filter on $\{\$a, \$v\}$, but there is no useful filter for just one of these parameters. We cannot use:

```
Answer(n) <- Type(p,$t)
```

to filter \$t, because the query is not safe (n appears in the head but not the body). However,

```
Answer(n) <- Buys(n,p) AND Type(p,$t)
```

is safe and may be used. A possible plan for optimizing this query flock is in Fig. 7. Figure 8 shows the preferred join order for the final step. □

```
OK1($a,$v) := FILTER({$a,$v}, Answer(n) <- Cust(n,$a,$v),
                     COUNT(Answer) >= s)
OK2($t) := FILTER({$t}, Answer(n) <- Buys(n,p) AND Type(p,$t),
                     COUNT(Answer) >= s)
OK($a,$v,$t) := FILTER({$a,$v,$t}, Answer(n) <- Cust(n,$a,$v), AND
                                   Buys(n,p) AND Type(p,$t) AND
                                   OK1($a,$v) AND OK2($t),
                     COUNT(Answer) >= s)
```

Figure 7: Query-flock plan for Example 4.2

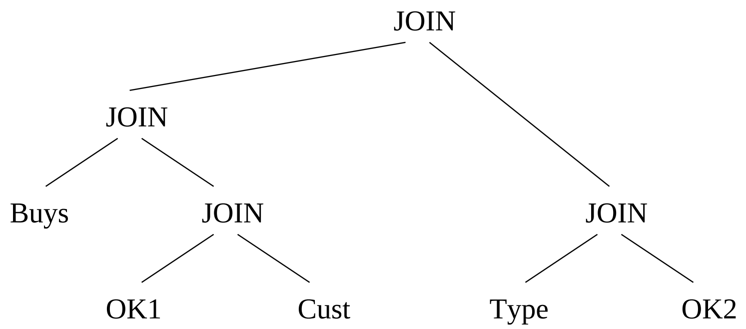

Figure 8: Join order for final step in Fig. 7