Query Flocks: A Generalization of Association-Rule Mining*

Dick Tsur, *Hitachi Corp.*
Jeffrey D. Ullman, *Stanford University*
Serge Abiteboul, *Stanford University and INRIA*
Chris Clifton, *MITRE Corp.*
Rajeev Motwani, *Stanford University*
Svetlozar Nestorov, *Stanford University*
Arnon Rosenthal, *MITRE Corp.*

October 29, 1997

Abstract

Association-rule mining has proved a highly successful technique for extracting useful information from very large databases. This success is attributed not only to the appropriateness of the objectives, but to the fact that a number of new query-optimization ideas, such as the "a-priori" trick, make association-rule mining run much faster than might be expected. In this paper we see that the same tricks can be extended to a much more general context, allowing efficient mining of very large databases for many different kinds of patterns. The general idea, called "query flocks," is a generate-and-test model for data-mining problems. We show how the idea can be used either in a general-purpose mining system or in a next generation of conventional query optimizers.

1 Introduction

We shall begin our discussion by reviewing the basics of market-basket analysis and the a-priori algorithm for finding items that tend to appear together in market baskets. We then see how to generalize the market-basket problem to "query flocks," that is, parametrized queries with a filter condition to eliminate values of the parameters that are "uninteresting." By expressing the query flock in Datalog, we find a well-known condition (query safety) that lets us enumerate the queries that are candidates for use in a query optimization technique that generalizes a-priori. Interestingly, the same technique, while applicable to SQL queries directly, is more apparent if we express queries in Datalog.

We then consider the pragmatics of implementing the generalized a-priori technique in a "query-flocks processor" or in a conventional SQL query optimizer. One approach is to see generalized a-priori as a cost-based optimization, principally involving join order and selection of some useful subqueries. Another approach is to view the technique as one that is applied dynamically, with the decision to perform an extra filtering step (analogous to the a-priori technique of eliminating low-support items) done only when we see the sizes of some intermediate relations during the query-flock execution process.

*This work was partially supported by the Community Management Staff's Massive Digital Data Systems Program, NSF grant IRI–96–31952, ARO grant DAAH04–95–1–0192, and grants of IBM and Hitachi Corp.

1.1 Review of Market-Basket Mining

The *market-basket* problem represents an attempt by a retail store to learn what items its customers frequently purchase together. The goal is an understanding of the behavior of typical customers as they navigate the aisles of the store. For instance, if we learn that customers frequently buy hamburgers and ketchup together, then we might suppose that many customers will walk from one to the other. If the store owner puts high-profit items tempting to such customers, e.g., relish, between, then they might induce more impulse buying and thus increase profits.

In the query problem, we are given a database containing information about "market baskets." That is, each time a customer appears at the cash register, the set of items they bought is entered in the database. We shall assume for simplicity that the database is a relation `baskets(BID,Item)`, giving pairs consisting of a basket "ID" and an item that appeared in that basket. The goal of market basket analysis is to find sets of items that are "associated," and the fact of their association is often called an *association rule*. Intuitively, associated items appear together frequently. Three more precise measures of association that have been used are:

1. *Support*: The items must appear in many baskets.
2. *Confidence*: The probability of one item given that the others are in the basket must be high.
3. *Interest*: That probability must be significantly higher or lower than the expected probability if items were purchased at random.

For example, an oft-repeated observation is that people who buy diapers often buy beer. The statement that the set $\{beer, diapers\}$ has high support means that many people buy both beer and diapers. That fact alone might be useful to a marketer. The statement that the rule $beer \rightarrow diapers$ has high confidence means that a lot of people who buy beer also buy diapers. That rule might be even more useful, although it begs the question whether people who buy beer are *especially* likely to buy diapers, or whether they buy diapers just because everybody buys diapers. Finally, saying that the rule $beer \rightarrow diapers$ has high interest means that if you buy beer, then you are much more (or much less) likely to buy diapers than the general population.

1.2 The A-Priori Optimization

There is an important trick for speeding up the search for high-support sets of items, known as *a-priori* ([AIS93], [AS94]). It uses the fact that if a set of items S appears in c baskets, then any subset of S appears in at least c baskets. For example, if we are looking for pairs of items that appear in at least c baskets, then we can start by finding those items that by themselves appear in at least c baskets. If c is high enough, we can eliminate most of the tuples in the `baskets` relation before we do the hard part: joining `baskets` with itself to count the occurrences of pairs of items. This transformation of the market-basket problem has been shown in the papers cited above to make a great difference in the time taken to find the answer to a question like "find all the pairs of items that appear in at least c market baskets."

1.3 The Problem With SQL as a Mining Language

In principle, we can express a query about pairs of items that appear in a large number of baskets in conventional SQL. This approach was examined by [HS95], for instance. The problem is that the right optimizations are beyond the state of the art in commercial database systems. For example, Fig. 1 shows how to express the query "find all pairs of items that appear together in at least 20 market baskets."[1] There, we join `baskets` with itself, with the condition that the basket ID's be the same, and the name of the first item be lexicographically less than the name of the second item (to avoid repeating pairs in both possible orders). We group the joined relation by the pair of items involved and check in the `HAVING` clause that the group has at least 20 baskets.

```
SELECT i1.Item, i2.Item
FROM baskets i1, baskets i2
WHERE i1.Item < i2.Item AND
      i1.BID = i2.BID
GROUP BY i1.Item, i2.Item
HAVING 20 <= COUNT(i1.BID)
```

Figure 1: Searching for association rules using SQL

The problem with this formulation is that the a-priori trick will not be implemented by conventional optimizers. For example, using a popular DBMS, we found that by rewriting the query of Fig. 1 to first find those items that appeared in at least 20 baskets (the data was actually word occurrences in newspaper articles, so the threshold of 20 occurrences made more sense than it would for retail-store data) and then joining the set of these items with the `baskets` relation before performing the query as written in Fig. 1, resulted in a 20-fold speedup.

In principle, the necessary code optimizations *could* be implemented in SQL systems. The idea is roughly "pushing grouping down the expression tree," and as such has been studied in the abstract by [GHQ95]. However, until the advent of market-basket mining, there has been little motivation for database vendors to invest a lot of time and effort in building these transformations into their systems. In this paper we argue the following:

- There are many data-mining problems besides market-basket analysis that could profit from building the a-priori form of code optimization into existing systems.
- There are a number of approaches to optimization of the a-priori type, including at least one that is significantly different from existing optimization techniques.
- The formalism of "query flocks," including their expression in Datalog, is an important tool for building improved optimizers.

1.4 Can We Mine in SQL?

It was pointed out in several early papers cited above that SQL systems are unable to compete with ad-hoc file processing algorithms such as a-priori and its variants. However, in this paper we assume that the data is stored in a conventional relational system and that mining occurs by issuing a sequence of SQL queries to the database. We cannot dispute the demonstrated fact that ad-hoc file processing algorithms can outperform, often significantly, DBMS-based algorithms. However, this fact does not negate the importance of the approach we are taking for two reasons:

1. The algorithms for mining and the optimizations we develop can be carried over to a file-based, rather than DBMS-based setting, with corresponding speedup.

[1] In practice market basket analysis is done with a much higher floor than 20 baskets, typically 1% of all baskets. We have used 20 throughout this paper as an example of a lower bound on support.

2. If mining of large-scale databases is ever to become a routine matter, where mining queries can be issued quickly to whatever data is appropriate, then DBMS's, probably SQL databases, must play an important role in this process. Even if it is found more appropriate to move the data out of a conventional DBMS into a special-purpose system designed for mining, then the algorithms we suggest, or improvements thereof, will have to be a component of the special-purpose system.

1.5 Outline of Paper

In Section 2 we introduce our model of a query flock and the running examples that we shall use to explicate the concepts. We select as our flocks language unions of conjunctive queries with arithmetic comparisons and negations allowed.

Section 3 discusses the ways in which the a-priori technique can be generalized to the setting of query flocks with filters based on minimum support. We show how the old concept of "safe conjunctive queries" guides the application of this technique in the general setting.

Then, in Section 4 we introduce a notation for query plans. The search for optimal plans is complicated by the fact that, unlike conventional optimization problems, there is not even an exponential space of possible plans to which we can restrict our search. We therefore suggest some reasonable heuristics for limiting the search to an exponential set of plans, including a "dynamic" technique where we select a join order in advance, but choose whether or not to apply a filter operation only after seeing an intermediate result.

2 Query Flocks

Intuitively, a *query flock* is a generate-and-test system, in which a family of queries that are identical except for the values of one or more "parameters" are asked simultaneously. The answers to these queries are filtered and those that pass the filter test enable *their parameters* to become part of the answer to the query flock. The setting for a query flock system is:

- A language in which we can express queries that are parametrized by one or more parameters.
- A language in which to express filter conditions about the results of a query.

Given these two languages, we can specify a particular query flock by designating:

1. One or more predicates that represent data stored as relations.
2. A set of *parameters*, which we shall always denote with names beginning with $.
3. A *query* expressed in our query language, using the parameters in roles normally reserved for constants.
4. A *filter* that specifies a condition that the result of the query must satisfy in order for a given assignment of values to the parameters to be acceptable.

The meaning of such a query flock is a set of tuples that represent the "acceptable" assignments of values for the parameters. We determine the acceptable parameter assignments by, in principle, trying all such assignments in the query, evaluating the query, and seeing whether the result passes the filter test. Of course there are often more efficient ways to compute the meaning of a query flock, and these optimizations are the subject of the balance of this paper.

- Remember: a query flock is a query about its parameters. The result of the flock is not the result of the parametrized query that is used to help specify the flock.

2.1 Our Languages for Flocks

The idea of expressing both a query form and a filter condition has been proposed before. For example, Mannila ([Man97]) talks about a logic in which both can be expressed. However, Mannila's formulation puts more in the filter, e.g., "one of the items in a market basket must be beer," while for us the role of the filter is limited to a condition about the result of the query. We would simply eliminate one of the parameters and mention beer explicitly in the query flock, should we require one of the items to be beer.

We shall use as our query language "conjunctive queries" [CM77], augmented with arithmetic (introduced in Section 2.4 and with union, as introduced in Section 3.4. In the following and subsequent examples, we shall use Datalog ([Ull88]) notation, rather than SQL, to express our queries. Datalog gives us two capabilities whose utility will become clear in Section 3:

1. The notion of "safe query" for Datalog figures into potential optimizations.
2. The set of options for adapting the a-priori trick to arbitrary flocks is most easily expressed in Datalog.

However, each of the advantages mentioned above can be translated to SQL terms.

For the filter language, we use SQL conditions, as might appear in a `HAVING` clause. The filter language turns out to be less important, since our principal results concern flocks for which the filter is a support condition, and such a condition is essentially a single constant, the minimum threshold for support.

2.2 Market Basket Analysis as a Query Flock

As our first example, we shall consider the simplest market-basket problem as a query flock. We are given a relation `baskets(BID,Item)` as underlying data, and we want to find those pairs of items `$1` and `$2` that appear in at least c baskets. This query flock easily generalizes to finding sets of k items that appear together for any fixed k.[2]

QUERY:

```
answer(B) :-
        baskets(B,$1) AND
        baskets(B,$2)
```

FILTER:

```
COUNT(answer.B) >= 20
```

Figure 2: Market basket association rules as a query flock

Example 2.1 : The query flock for finding pairs of items that appear in at least 20 baskets is seen in Fig. 2. For any values of `$1` and `$2`, the query asks for the set of baskets B in which items `$1` and `$2` both appear. The answer relation for this pair of items is the set of such baskets. Then, the

[2] However, finding something more complex, like the set of maximal sets of items that appear in at least c baskets (regardless of the cardinality of the set of items), is more awkward and would be expressed as a sequence of query flocks for increasing cardinalities, with each flock depending on the result of the previous flock.

filter condition requires that the set of such baskets number at least 20. The result of the query flock is thus the set of pairs of items (`$1`, `$2`) such that there are at least 20 baskets containing both items `$1` and `$2`. □

2.3 Representing Confidence in Query Flocks

The most natural query flocks, and indeed the flocks for which we have the most promising optimization techniques, involve support as the filter condition; Example 2.1 is such a flock. It is possible to represent confidence, interest, and other conditions as filters, using SQL-like conditions involving aggregation. However, it is necessary to allow the query portion of a flock to produce several relations as a result.

```
QUERY:

    answer1(B) :-
            baskets(B,$1) AND
            baskets(B,$2)

    answer2(B) :-
            baskets(B,$1)

FILTER:

    2 * COUNT(answer2.B) >= COUNT(answer1.B)
```

Figure 3: A flock that asks for high confidence

Example 2.2 : Suppose we want to find pairs of items `$1` and `$2` such that the confidence of `$2` given `$1` is at least 50%. The flock can be written as in Fig. 3. Here, there are two answer relations, which we call `answer1` and `answer2`. The first counts the number of baskets containing both items, while the second counts the number of baskets containing the first item. The condition asks that the ratio of these counts be at least 1/2. □

2.4 Adding Arithmetic and Negation to Our Query-Flocks Language

The market-basket problem corresponds to a very simple query flock. The query is a conjunctive query whose only subgoals are positive, relational subgoals. In principle, any query language whatsoever could be used as the query flock language. However, in order to apply the query optimization techniques we propose, there are some limitations on the query language. The extensions to conjunctive queries that we shall allow are:

1. Negated subgoals.
2. Arithmetic subgoals, e.g., $X < Y$, where X and Y are variables or parameters.

We shall refer to this broader class of conjunctive queries as *extended conjunctive queries*. In addition, we shall allow a query that is the union of these extended CQ's, which we discuss in Section 3.4. However, as with the original CQ's, we assume that extended CQ's follow the conventional *set semantics* rather than bag semantics, where duplicate tuples are allowed. Some of our claims would not hold for bag semantics.

As a simple example of where arithmetic subgoals are useful, the original query flock for market baskets, Example 2.1, produces each successful pair of items in two orders. We can restrict the result to have each pair of items appear only in lexicographic order if we add an arithmetic condition to the query, as:

```
answer(B) :- baskets(B,$1) AND baskets(B,$2) AND $1 < $2
```

The other two extensions (negation and union) are also quite useful. We introduce them in the following two examples, which we use throughout the rest of the paper to illustrate our ideas.

Example 2.3: The following is an example of a query flock that searches for unexplained side-effects. That is, we want to find symptoms `$s` and medicines `$m` such that there are many patients (and as before, we take 20 to be the threshold of "many") that exhibit the symptom and are taking the medicine, yet the patient's disease does not explain the symptom. The underlying data with which we work consists of the following relations:

1. `diagnoses(Patient, Disease)`: The patient has been diagnosed as having the disease.
2. `exhibits(Patient, Symptom)`: The patient exhibits the symptom.
3. `treatments(Patient, Medicine)`: The medicine has been prescribed for the patient.
4. `causes(Disease, Symptom)`: The disease is known to cause the symptom.

The query flock for the problem described above is shown in Fig. 4. In order for this query to make sense, we assume that each patient has one disease only. To include patients with several diseases simultaneously, we would have to extend our query-flocks language to allow intermediate predicates (in particular, a predicate relating patients to the set of symptoms from all their diseases). That extension is feasible but we shall concentrate on the simpler cases in order to explore their query-optimization opportunities.

QUERY:

```
    answer(P) :-
            exhibits(P,$s) AND
            treatments(P,$m) AND
            diagnoses(P,D) AND
            NOT causes(D,$s)
```

FILTER:

```
    COUNT(answer.P) >= 20
```

Figure 4: Mining for side-effects in a medical database

In the flock of Fig. 4, the parametrized query asks for the set of patients P that exhibit a symptom `$s`, are receiving medicine `$m`, have disease D, and yet the disease D doesn't explain the symptom `$s`. The filter requires that there be at least 20 patients taking medicine `$m` and exhibiting unexplained symptom `$s`. □

Example 2.4: In our next example, we are looking for words that are strongly related in a collection of HTML documents. While there are many notions of "strong connection" that could be explored, we shall fix on one, in which we count

1. The number of times the words appear together in a title, and
2. The number of times one word appears in an anchor and the other appears in the title of the document the anchor points to.

The query flock that searches for such pairs of words is based on the following predicates or relations:

1. `inTitle(D,W)`: Word W is in the title of document D.
2. `inAnchor(A,W)`: Word W appears in the anchor text of anchor A.
3. `link(A,D1,D2)`: Anchor A links document $D1$ to document $D2$.

QUERY:

```
answer(D) :-
        inTitle(D,$1) AND
        inTitle(D,$2) AND
        $1 < $2

answer(A) :-
        link(A,D1,D2) AND
        inAnchor(A,$1) AND
        inTitle(D2,$2) AND
        $1 < $2

answer(A) :-
        link(A,D1,D2) AND
        inAnchor(A,$2) AND
        inTitle(D2,$1) AND
        $1 < $2
```

FILTER:

```
COUNT(answer(*)) >= 20
```

Figure 5: A query flock defining strongly connected words

Figure 5 shows the query flock that finds pairs of words in the two desired relationships: together in title, or one in anchor and the other in target title. To prevent pairs of words from being generated twice, we have elected to require that the first word, `$1` lexically precede the second word, `$2`. As a result, we are forced to use three rules, since we must distinguish the case where the lexically first word is in the anchor from the case where the second word is in the anchor.

Again we have taken 20 occurrences as the threshold of significance. Notice that the count in the filter is counting answers, which may be either anchor ID's or document ID's. We assume that there are no values in common between these two types of ID's, or the count could be too low. □

3 Generalizing the A Priori Technique

The essence of the a-priori trick applied to query flocks is that we optimize by first evaluating a less expensive query whose answer allows us to upper bound the size of the answer that would be obtained with certain parameters. If that bound is less than the threshold in the filter condition, we can eliminate certain values of a parameter or parameters without further consideration. But where do these less expensive queries come from?

3.1 Containment for Conjunctive Queries

When the queries involved are conjunctive ([CM77], [Ull89], [AHV95]), there is a straightforward answer to the question. The simplest way for a query Q_1 to put an upper bound on the size of the result of a query Q_2 is for it to be provable that for any database, the result of Q_2 is a subset of the result of Q_1, a condition which we normally write $Q_2 \subseteq Q_1$. However, for conjunctive queries, this containment is decidable, using the technique of containment mappings ([CM77]). A consequence of the containment-mapping theorem is that the only way $Q_2 \subseteq Q_1$ can hold is if Q_1 is constructed from Q_2 by

1. Taking a subset of the subgoals of Q_2, and
2. Splitting zero or more variables into several variables.

Splitting variables can neither decrease the number of solutions to the query nor make the query simpler. Thus, we shall limit our search to subsets of the subgoals of Q_2, with no variable splitting allowed. Picking a proper subset of the subgoals also does not decrease the size of the solution, but it can make the query simpler, and that simplification is the essence of the a-priori trick. Using a subset of the subgoals can also eliminate from consideration some of the parameters of the query flock, another important aspect of "a-priori."

3.2 Safe Queries

However, not every subset of the subgoals of a conjunctive query makes sense as an intermediate step in the evaluation of the query flock. In particular, if the variables that appear in the head of the query do not also appear in the body, then the query defines an infinite set of tuples for the head predicate, and therefore could not provide a useful upper bound on the size of the result for the full query. This condition has been studied before ([Ull88]) as a way to restrict Datalog queries to be equivalent to relational algebra; it is called *safety*, and CQ's (without negation or arithmetic) that satisfy the condition

Each variable that appears in the head also appears in the body

are called *safe queries*.

Example 3.1: Consider the market-basket query flock from Example 2.1, which we reproduce here:

```
answer(B) :-
        baskets(B,$1) AND
        baskets(B,$2)
```

There are only two nontrivial subqueries formed by taking a nonempty, proper subset of the subgoals,

```
answer(B) :- baskets(B,$1)
```

and

```
answer(B) :- baskets(B,$2)
```

Either can be used to prune values of one of the parameters. For instance, if we use the first, then we can ask for what values of `$1` does the query `answer(B) :- baskets(B,$1)` produce a number of values of B that is over the threshold given in the filter. Any other value of `$1` can be eliminated from consideration as a member of a pair of items meeting the filter condition.

By symmetry, the set of `$1`'s that survive a test based on the first subquery is exactly the same as the set of `$2`'s that will survive a test based on the second subquery. In fact, the a-priori trick (at least for the case of item pairs) can be seen as the combination of the use of one of these two subqueries and the exploitation of their equivalence. □

We may summarize the generalization of a-priori for CQ's without negation or arithmetic, as follows:

> **Optimization Principle for Conjunctive Queries:** To optimize a query flock described as a conjunctive query Q and a filter that puts a lower bound s on support, consider evaluating only those safe subqueries formed by deleting one or more subgoals from Q. Exploit such a query by eliminating values of the parameters that do not meet the support threshold s when that subquery is evaluated.

3.3 Safe Queries with Negation and Arithmetic

When we expand our horizon beyond conjunctive queries to the Datalog queries with negation and arithmetic that we have been using, matters get more complex in several ways. First, the discovery of containing queries is not as easy. There are decision procedures — [Klu82] or [ZO93] for Datalog with arithmetic, and [LS93] for Datalog with negation, including arithmetic. However, there are some cases where the containing query cannot be characterized as a subset of the subgoals of the contained query.

Since these cases are unusual, we propose to continue our restriction that we look only at subsets of the subgoals of the query that defines the query flock. We then have only to augment our search with the generalized notion of what a safe query is. There are now three conditions that must be satisfied ([UW97]):

1. Every variable that appears in the head must appear in a nonnegated, nonarithmetic subgoal of the body.
2. Every variable that appears in a negated subgoal of the body must appear in a nonnegated, nonarithmetic subgoal of the body.
3. Every variable that appears in an arithmetic subgoal of the body must appear in a nonnegated, nonarithmetic subgoal of the body.

However, parameters are *variables*, not constants, as far as the above safety conditions are concerned. Since they cannot appear in the head, they are not affected by rule (1). However, the last two rules apply to parameters as well as to explicit variables.

Example 3.2: Let us consider the flock of Example 2.3, which includes a negated subgoal; we repeat it here:

```
answer(P) :-
        exhibits(P,$s) AND
        treatments(P,$m) AND
        diagnoses(P,D) AND
        NOT causes(D,$s)
```

Which of the 14 nontrivial subsets of the subgoals are safe? First, to satisfy condition (1), one of the subgoals must include the head variable P. That condition rules out only one possible subquery:

```
answer(P) :- NOT causes(D,$s)
```

Notice that this query makes no sense, since it is trying to count a number of patients, but the only information we have to go on says that some disease D does not cause the symptom `$s`.

We also must assure condition (2), which says that if we pick the subgoal `NOT causes(D,$s)`, then since variable D and parameter `$s` appear in this subgoal, we must also pick a positive subgoal that has D in it and a positive subgoal that has `$s` in it. That is, if we pick `NOT causes(D,$s)`, then we must also pick both `diagnoses(P,D)` and `exhibits(P,$s)`, the only positive subgoals with D and `$s`, respectively. Thus, condition (2) again rules out the subquery above that has only `NOT causes(D,$s)` in its body and also rules out the other five subqueries that have this subgoal but do not have both of `exhibits(P,$s)` and `diagnoses(P,D)`.

The remaining eight subqueries are candidates for use in an optimization where we use the subquery to eliminate values for `$s`, `$m`, or perhaps (`$s`, `$m`) pairs, before we evaluate the entire query flock. Before considering the optimization problem in general, let us consider some likely candidates. In their interpretation, recall that the filter condition for the original query flock is that there must be at least 20 patients receiving the medicine `$m` and exhibiting the unexplained symptom `$s`.

1. `answer(P) :- exhibits(P,$s)`. At least 20 patients exhibit the symptom.
2. `answer(P) :- treatments(P,$m)`. At least 20 patients must have been given the medicine.
3. `answer(P) :- diagnoses(P,D) AND exhibits(P,$s) AND NOT causes(D,$s)`. There are at least 20 patients with a disease that does not cause a symptom they exhibit.
4. `answer(P) :- exhibits(P,$s) AND treatments(P,$m)`. There are at least 20 patients taking the medicine and exhibiting the symptom.

We cannot pick a strategy without knowing something about sizes of the relations and numbers of patients, diseases, etc. However, there are some intuitive observations we can make.

- Either (1) or (3) could be used as a preliminary filter for `$s` values, or both could be used, with (1) used before (3). Which of these three options, or none, makes sense depends on the statistics of the situation. For example, (1) is attractive only if there are many `exhibits` tuples for rare symptoms. Subquery (3), which is almost the entire query except for the introduction of medicines, is attractive if the number of different medicines administered for a disease is small; then a symptom that qualifies under (3) is very likely to be associated with a single medicine that will meet the threshold of 20 patients.
- (1) and (2) may both be useful subqueries. Subquery (1) can be used to eliminate rare symptoms from consideration, and (2) can be used to eliminate rarely used medicines. However, whether it is worth basing a preliminary step on (1) and/or (2) depends on the density of rare symptoms and medicines. For example, if threshold 20 patients were such that only 1/10 of

the symptom reports were for symptoms that appeared in 20 or more patients, then we could get significant advantage if we eliminated from relation `exhibits` all those tuples with rare symptoms. Conversely, if almost all symptom reports were for symptoms appearing in at least 20 (or whatever the support threshold were) patients, then subquery (1) would not be worth the extra effort.

- Even though subquery (4) involves both `$s` and `$m`, it might be a useful preliminary step. It might be easier to join the two relations `exhibits` and `treatments` than to join all four relations that appear in the full query. It is then likely that there are lots of symptom-medicine pairs that do not appear in 20 patients, and these can be eliminated from consideration before we join in `symptoms` and `causes` to determine which of the frequently occurring symptom-medicine pairs are explained by the fact that the medicine is used to treat a disease that causes the symptom.

□

3.4 Extension to Unions of Datalog Queries

Suppose a query flock consists of a union of Datalog queries of the type that we have been considering. We can construct a query that provides an upper bound on the result of the union if we take the union of queries that provide an upper bound on each query individually. Thus, we must look for a subquery for each query in the union. Each query must be safe, in the sense described in Section 3.2. If so, then the size of the result of the union of the subqueries will be a bound on the size of the result for the original query. We may thus use the union of subqueries to eliminate values of a parameter or parameters that cannot possibly appear in the result of the query flock.

> **Optimization Principle for Unions of Conjunctive Queries:** To optimize a query flock described as a union of conjunctive queries $Q_1, Q_2, \ldots, Q_n$ and a filter that puts a lower bound s on support, consider evaluating only those unions of safe subqueries $P_1, P_2, \ldots, P_n$ such that P_i is formed by deleting one or more subgoals from Q_i, for $i = 1, 2, \ldots, n$. Exploit such a union of queries by eliminating values of the parameters that do not meet the support threshold s when that subquery is evaluated.

Example 3.3: Let us consider the union in Example 2.4, and suppose we want to find a subquery that involves only word `$1`. Because of the safety condition, there is essentially only one choice for each of the three queries in the union, and these subqueries are:

```
answer(D) :- inTitle(D,$1)
answer(A) :- inAnchor(A,$1)
answer(A) :- link(A,D1,D2) AND inTitle(D2,$1)
```

That is, a word cannot be a candidate for `$1` (or for `$2` for that matter) unless we get to at least 20 when we sum the:

1. Number of times it appears in a title,
2. Number of times it appears in an anchor, and
3. Number of anchors that point to titles in which it appears.

□

4 Search for Optimal Query-Flock Evaluators

We have limited our search for evaluation strategies to the selection of some subqueries that we use to restrict the values of one or more parameters. In this section we introduce a formal notation for the use of such subqueries and use them to represent query plans.

4.1 Filter Steps

Let us use the expression

```
R(P) := FILTER(P,Q,C)
```

where

1. P is a set of parameters,
2. Q is a query involving the parameters P,
3. R is a relation whose tuples are values of the parameters P, and
4. C is a condition on the result of the query Q

to mean:

> Create relation R to consist of one tuple for each assignment of values for the parameters P such that with those parameter values the result of query Q meets the condition C.

A *query plan* is a sequence of filter steps. Each step can use in subgoals any of the relations that hold the data of the problem and any of the relations about the parameters that were created by previous steps.

Example 4.1 : Let us continue Example 3.2. Suppose that, using some estimate for the expected sizes of relations and joins, we conclude that the best strategy for finding unexpected side-effects is to filter the symptoms using subquery 1 (at least 20 patients exhibit the symptom) and also to filter medicines using subquery 2 (at least 20 patients are taking the medicine, but not to filter symptoms by subquery 3 or filter symptom-medicine pairs by subquery 4. Then our query plan consists of three steps, as shown in Fig. 6:

1. Create a unary relation `okS` consisting of all those symptoms that appear in at least 20 patients.
2. Create a unary relation `okM` consisting of all those medicines that are given to at least 20 patients.
3. Evaluate the entire query, using the original four subgoals plus additional subgoals that are the `okS` and `okM` relations.

In the first step of Fig. 6 we define the relation `okS`. The set of parameters is `$s` alone. The query for the `FILTER` step is the subquery that we suggested in Example 3.2 would be useful for filtering out rarely occurring symptoms. The filter condition in this step, as in all three steps, is the condition that the result of the query contain at least 20 patients. The second step is a similar and creates the set `okM` of medicines that are used on at least 20 patients.

Then, the third step repeats the original query of Example 3.2, but now there are two additional subgoals, `okS($s)` and `okM($m)`. One might ask: have we not made things harder? In addition to the first two `FILTER` steps, which take some time, we finish the query plan with a step that is the original query flock plus two extra subgoals.

```
okS($s)   := FILTER($s,
                 answer(P) :- exhibits(P,$s),
                 COUNT(answer.P) >= 20
             );

okM($m)   := FILTER($m,
                 answer(P) :- treatments(P,$m),
                 COUNT(answer.P) >= 20
             );

ok($s,$m) := FILTER({$s,$m},
                 answer(P) :-
                     okS($s) AND
                     okM($m) AND
                     diagnoses(P,D) AND
                     exhibits(P,$s) AND
                     treatments(P,$m) AND
                     NOT causes(D,$s),
                 COUNT(answer.P) >= 20
             );
```

Figure 6: A query plan for the medical mining problem

However, the third step should be easier, not harder, to answer than the original query. The intuitive reason is that the subgoals `okS($s)` and `okM($m)` can be joined with other subgoals — `exhibits(P,$s)` and `treatments(P,$m)` in particular — relatively quickly. Moreover, the results of these joins will be smaller relations, thus making subsequent join steps take less time than they would in the original query flock. □

4.2 Legal Query Plans

One can argue intuitively that the query plan expressed in Example 4.1 meets its most basic requirement — the result of the sequence of filter steps is equivalent to the original query flock, which expressed as a single filter step is:

```
ok($s,$m) := FILTER({$s,$m},
                 answer(P) :-
                     diagnoses(P,D) AND
                     exhibits(P,$s) AND
                     treatments(P,$m) AND
                     NOT causes(D,$s),
                 COUNT(answer.P) >= 20
             );
```

One could ask under what conditions the plan of Fig. 6 is an improvement on the original, and if so, what join order should be used for the final step. We cannot give a definitive answer to such questions without estimates for sizes of join results, but the matter has been studied extensively, and the general theory of cost-based optimization ([G*79], e.g.) applies here.

However, there is a more fundamental question that must be settled: under what circumstances is a query plan equivalent to a given query flock? We propose the following as a natural consequence of the ideas presented so far. First, we treat only filters that involve support; i.e., the filter condition is a lower bound on the size of the query result. Handling other filters remains open. Possibly there are more general rules that could be proposed, even for support-type filters, but we believe that the space of options implied by the following rule will omit the best option only in pathological cases.

> **Rule for Generating Query Plans for Conjunctive Query Flocks with Support-Type Filter Conditions:** Consider only sequences of `FILTER` steps that meet all of the following conditions:
>
> 1. Each step uses the same filter condition as the original query flock.
> 2. Each step defines a uniquely named relation.
> 3. Each step is derived from the given query flock by the following steps:
>   (a) Start with the original query flock.
>   (b) Add in zero or more subgoals that are copies of the left side of the assignment (`:=`) in some previous filter step.
>   (c) Delete zero or more subgoals but, following the optimization principle for conjunctive queries, make sure that the resulting query is safe.
> 4. The final step must not delete any subgoals of the original query; it may have additional subgoals derived from previous steps, of course.

Example 4.2 : In Fig. 6, each of the three steps use the same filter condition as the original query flock. The first two steps do not add any additional subgoals to the query, but they delete all but one of the original subgoals. The result is a safe query in each case. The last step retains all four subgoals of the original query flock, and adds to it the left sides of the first two steps. Notice that these left sides must be copied literally, using the same relation name as the predicate and the same parameters as arguments. □

4.3 An Exponential Search for Query Plans

There is ample precedent for making exponential searches to find the best query plan, for instance [G*79]. Because queries tend to be small, exponential searches are often computationally feasible. However, the space of query plans entailed by the rule in Section 4.2 is not bounded by an exponential in the size of the query defining the original query flock. Although the number of safe subsets of the subgoals of the original query is no more than exponential, there is also the option of adding subgoals that are the left sides of prior queries. Thus, each time we add a step to our query plan, we double the number of options for the next step.

Moreover, there is some reason to believe that a long sequence of steps, in which each uses the result of the previous step, is at least a candidate for being best-possible. The following is an example that illustrates the point.

Example 4.3 : Suppose our query flock is based on an underlying relation `arc` that represents arcs of a directed graph. This query appears in Fig. 7. Intuitively, the query flock asks about a node `$1` whether it has at least 20 successors from which there is a path of length n extending.

QUERY:

```
answer(X) :- arc($1,X) AND arc(X,Y_1) AND arc(Y_1,Y_2) AND ...  AND arc(Y_{n-1},Y_n)
```

FILTER:

```
COUNT(answer.X) >= 20
```

Figure 7: A pathological query flock

The first step of a query plan might use only the first subgoal, i.e., restrict ourselves to nodes `$1` that have at least 20 successors, regardless of what paths extend from these. A second step might examine the nodes that past the first test and, with the first two subgoals of the original query restrict to nodes that have at least 20 successors that themselves have successors. We can proceed in this fashion, winding up with an $n+1$-step query plan, any step of which might make a useful simplification of the query. The pattern of this query plan is suggested by Fig. 8. □

```
ok0($1) := FILTER($1,
                  answer(X) :- arc($1,X),
                  COUNT(answer.X) >= 20
           );

ok1($1) := FILTER($1,
                  answer(X) :- ok0($1) AND arc($1,X) AND arc(X,Y1),
                  COUNT(answer.X) >= 20
           );

...

okn($1) := FILTER($1,
                  answer(X) :- ok[n-1]($1) AND arc($1,X) AND
                               arc(X,Y1) AND ... AND arc(Y[n-1],Yn),
                  COUNT(answer.X) >= 20
           );
```

Figure 8: A query plan with $n+1$ steps

There are many reasonable ways that we could restrict the search for a query plan to an exponential number of possibilities. Here are two that can be said to generalize the a-priori technique:

1. Select some sets of parameters. For each selected set S, select a subset of the subgoals of the original query that is safe and includes exactly the parameters of S. Use this subquery to define a relation R_S that restricts the parameters S. Finally, at the last step, use the original query together with all the subgoals formed from the relations R_S for each selected set of

parameters S. This approach generalizes a-priori for the case of two-item sets, and it is also followed by the query plan of Fig. 6, for instance.

2. Select a list of subsets of the subgoals of the original query that form safe queries. Turn each subquery Q into a `FILTER` step, first adding to Q any subgoals that can be formed from the result of a prior step that restricts parameters that appear in Q. This approach would yield the a-priori method for sets of more than two items. In that case, we compute candidate sets of k items by restricting to those itemsets such that each subset of $k-1$ items previously has met the support test.[3]

4.4 Dynamic Selection of Filter Steps

While the above strategies for limiting search to an exponential number of possible query plans are unremarkable, there is another strategy that has no analog in conventional query optimization. Instead of deciding on subqueries in advance, we let the sizes of intermediate relations *after we compute them* determine whether or not to apply a filter step. Intuitively, if the size of an intermediate relation is such that the average number of tuples per assignment of values to the parameters is significantly lower than it was at any previous step that computed a relation with the same set of parameters, then there is a good chance that many value-assignments for the parameters will be eliminated on this step, even though they were not eliminated previously.

There is one important special case: when the set of parameters for a relation has not previously been encountered. This case includes expressions consisting of a single subgoal, i.e., one of the relations upon which the query flock is based, provided that subgoal has one or more parameters among its arguments. For this special case, where there has been no previous filtering, we should ask whether the number of tuples per value-assignment for the parameters is low or high compared with the support threshold.

- If low, then we expect a lot of value-assignments to be eliminated, and it is likely to be useful to filter.
- If high, then few value-assignments are likely to be eliminated, and even if they are, their elimination will not reduce the size of the underlying relation significantly, so we should not filter at this point.

An example should help clarify the technique.

Example 4.4 : Let us again consider the query flock of Example 3.2, which searches for unexplained side-effects. We start by choosing a join order for the four subgoals. Any of a number of models and approaches to selecting this join order may be used; our idea is independent of how the join order is actually chosen. We shall suppose for argument that the ordering of Fig. 9 is chosen.

We start with the leaves of the tree of Fig. 9. Before we use leaf `exhibits(P,$s)`, we have the option to apply a `FILTER` step on `$s` that is equivalent to the computation of relation `okS` in Fig. 6. Whether we should do so depends on how the number of patients mentioned in `exhibits` compares with the number of symptoms mentioned there. Assuming this ratio is somewhat below 20 (our usual assumed threshold of support), then we shall elect to perform the filtering step. Note that because we eliminate the rare symptoms, we would normally want the ratio to be significantly less than 20 in order for this step to, say, reduce the size of the relation by half. However, since the actual distribution of the sizes of the groups for each symptom affects our expected reduction in

[3] The a-priori method takes advantage of symmetry among the parameters that represent items, making this process simpler than it would be in the general case.

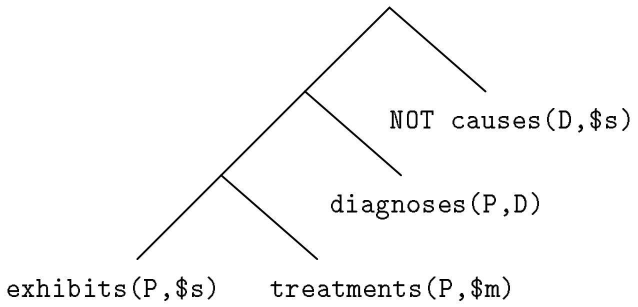

Figure 9: A join order for the medical-mining example

the relation size, we may want to do substantial gathering of statistics to support the filter/don't filter decision.

We also consider the leaf `treatments(P,$m)`. Assuming that there are more than 20 patients per medicine, we may decide that filtering `$m` at this time is likely to be unproductive. We cannot filter the leaf `diagnoses(P,D)`, because there are no parameters present. We also cannot filter leaf `NOT causes(D,$s)`, because the query with just this subgoal is not safe.

Now, consider the lowest interior node of the tree in Fig. 9. It represents the join

$$\texttt{exhibits(P,\$s)} \bowtie \texttt{treatments(P,\$m)}$$

and therefore involves both parameters. We have not seen a lower node involving both parameters, so we again are in the special case where we must decide whether the number of $(\$s, \$m)$ pairs eliminated is worth the cost of filtering. That is, we divide the size of the intermediate relation computed at that node by the product of the number of symptoms and medicines and compare this number with 20. Let us suppose that the ratio is low enough that we decide filtering is a good idea.

We then move to the node above, where `diagnoses(P,D)` is joined in. Assuming that patients appearing in `exhibits` and `treatments` also appear in `diagnoses`, the result of the second join cannot be smaller than the first, so there will not be any advantage to another `FILTER` step. However, we do not have to make this decision until after we join.

The final step is the join with `NOT causes(D,$s)` that completes the query. We must filter at the root, simply because that filtering is necessary to find the answer to the query flock. The resulting query plan looks a little different from the pure `FILTER` programs we have discussed previously, since the joins are performed explicitly. The plan appears in Fig. 10. □

5 Conclusions

We have presented a notation, called "query flocks," for describing large-scale data-mining operations. A flock consists of a parametrized query and a filter that selects certain assignments of values for the parameters by applying a condition to the result of the query for that value assignment. We have explored the case where the query is described by a union of one or more conjunctive queries with optional arithmetic and negation, and the filter is a lower bound on the number of tuples returned by the query.

We generalized the well-known a-priori technique for market-basket analysis to apply to any query flock in our class. By using the concept of query safety, we described the possible subqueries

```
temp1($s) := FILTER($s,
                    answer(P) :- exhibits(P.$s),
                    COUNT(answer.X) >= 20
                   )

temp2(P,$s,$m) := (temp1($s) JOIN exhibits(P,$s)) JOIN treatments(P,$m)

temp3($s,$m) := FILTER({$s,$m},
                       answer(P) :- temp2($s,$m).,
                       COUNT(answer.X) >= 20
                      )

temp4(P,D,$s,$m) := ((temp3($s,$m) JOIN temp2(P,$s,$m))
                         JOIN diagnoses(P,D)) JOIN (NOT causes(D,$s)

sideEffect($s,$m) := FILTER({$s,$m},
                            answer(P) :- temp4(P,D,$s,$m),
                            COUNT(answer.X) >= 20
                           )
```

Figure 10: A possible query plan resulting from dynamic evaluation

that could be used to exploit the a-priori idea, and we then suggested several techniques for further limiting the search for query plans. These techniques are either static heuristics, where we enumerate a class of plans and estimate the cost of each, based on available size estimates for relations, or dynamic, where we see the size of intermediate results before deciding whether or not to apply a filtering step.

Future Work

We can express many interesting patterns in the query flock framework described in this paper. There are, however, many more interesting, albeit more complex, patterns that cannot be expressed directly. Currently, we are investigating several different ways of extending the query flock framework in order to handle more complex patterns.

Extension to Datalog Programs: In this paper we considered query flocks consisting of a single CQ or a union of CQs. The next natural extension is to consider some restricted form of Datalog programs. As a first step we can allow auxiliary predicates defined as CQ over the base relations. Then we can use these predicates in the query flock.

Monotone Filter Conditions: The techniques described in this paper apply directly to any *monotone* filter condition. By monotone we mean that if the condition is true for a given set then it must also be true for any superset of the original set. Examples include certain COUNT, MIN, MAX, SUM (in the case of non-negative numbers) conditions. As a simple example, we can extend the traditional market basket problem, whose flock appeared in Fig. 2 to a *weighted market basket*, where the baskets B have weights W associated through a relation `importance(B,W)`. For example, in conventional market-basket mining, the importance of a basket might be the total value of all items purchased, or in test mining, the baskets could be documents, the items could be words, and the importance of a document is the number of web hits it gets. In any event, we can modify the original market-basket flock to evaluate an answer by summing the weights of the baskets returned in the answer, as in Fig. 11.

```
QUERY:

     answer(B,W) :-
             baskets(B,$1) AND
             baskets(B,$2) AND
             importance(B,W)

FILTER:

     SUM(answer.W) >= 20
```

Figure 11: Weighted market basket, an example of a monotone flock

Extending Filter Conditions: There are also many nonmonotone filter conditions that are of interest, including those that involve looking for high degrees of correlations, confidence, or other statistical properties. Another problem occurs when the filter condition does not rule out infrequently occurring values. For example, in text mining, we are often interested in medium numbers of occurrences of word pairs, since very common words are uninteresting, but a very few co-ocurrences of a word pair might be statistically insignificant. The methods proposed so far do not work well when the threshold of occurrences is not a fairly strong lower bound. Are there new methods that can work for these and other nonmonotone filters?

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, Reading, Mass., 1995.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," *Proc. ACM SIGMOD Conf.*, pp. 207–216, 1993.

[AS94] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," *Proc. 20th VLDB Conf.*, 1994.

[CM77] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational databases," *Proc. Ninth Annual ACM Symposium on the Theory of Computing*, pp. 77–90.

[G*79] P. P. Griffiths (Selinger), M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price [1979]. "Access path selection in a relational database management system," *ACM SIGMOD International Conf. on Management of Data*, pp. 23–34., 1979.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass, Generalized projections, a powerful approach to aggregation, *Proc. 21st VLDB Conf.*, 1995.

[HS95] M. Houtsma and A. Swami, "Set-oriented mining of association rules," *Proc. Intl. Conf. on Data Engineering*, pp. 25–34.

[Klu82] A. Klug, "Equivalence of relational algebra and relational calculus query languages having aggregate functions," *J. ACM* **29**:3, pp. 699–717.

[LS93] A. Y. Levy and Y. Sagiv, "Queries independent of update," *Proc. International Conference on Very Large Data Bases*, pp. 171–181, 1993.

[Man97] H. Mannila, "Methods and problems in data mining," *Proc. Intl. Conf. on Database Theory*, 1997, pp. 41–55, Springer-Verlag.

[Ull88] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Volume I—Fundamental Concepts, Computer Science Press, New York., 1988.

[Ull89] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Volume II—The New Technologies, Computer Science Press, New York., 1989.

[UW97] J. D. Ullman and J. Widom, *A First Course in Database Systems*, Addison-Wesley, Reading, Mass., 1997.

[ZO93] X. Zhang and M. Z. Ozsoyoglu [1993]. "On efficient reasoning with implication constraints," *Proc. Third DOOD Conference*, pp. 236–252, 1993.