Integrating Data Mining with Relational DBMS: A Tightly-Coupled Approach

Svetlozar Nestorov[1] and Shalom Tsur[2*]

[1] Department of Computer Science, Stanford University,
Stanford, CA 94305, USA
evtimov@db.stanford.edu
http://www-db.stanford.edu/people/evtimov.html
[2] Surromed, Inc.
Palo Alto, CA 94303, USA
tsur@surromed.com

Abstract. Data mining is rapidly finding its way into mainstream computing. The development of generic methods such as itemset counting has opened the area to academic inquiry and has resulted in a large harvest of research results. While the mined datasets are often in relational format, most mining systems do not use relational DBMS. Thus, they miss the opportunity to leverage the database technology developed in the last couple of decades.

In this paper, we propose a data mining architecture, based on the query flock framework, that is tightly-coupled with RDBMS. To achieve optimal performance we transform a complex data mining query into a sequence of simpler queries that can be executed efficiently at the DBMS. We present a class of levelwise algorithms that generate such transformations for a large class of data mining queries. We also present some experimental results that validate the viability of our approach.

1 Introduction

Data mining — the application of methods to analyze very large volumes of data so as to infer new knowledge — is rapidly finding its way into mainstream computing and becoming commonplace in such environments as finance and retail, in which large volumes of cash register data are routinely analyzed for user buying patterns of goods, shopping habits of individual users, efficiency of marketing strategies for services and other information. The development of generic methods such as itemset counting and the derivation of association rules [1,3] has opened the area to academic inquiry and has resulted in a large harvest of research results.

From an architectural perspective, the common way of implementing a data mining task is to perform it using a special purpose algorithm which typically analyzes the data by performing multiple sequential passes over a data file. The

* This work was done while the author was affiliated with Research and Development Lab, Hitachi America, Ltd., Santa Clara, California, USA.

performance measure is usually the number of passes required to conclude the analysis. There are obvious advantages in integrating a database system in this process. In addition to such controlling parameters as support and confidence levels, the user has an additional degree of freedom in the choice of the data set to be analyzed, which can be generated as the result of a query. Furthermore, the well understood methods for query optimization, built into the DBMS, can be utilized without further development. While the potential benefits of an integrated data-mining/DBMS system are easy to perceive, there is a performance issue that requires consideration: can we achieve a comparable, or at least an acceptable level of performance from these integrated methods when compared to the special-purpose external methods? This question was previously examined in a more narrow context of association rules and a particular DBMS in [7] and [2]. Section 2 of this paper elaborates on the general architectural choices available and their comparison.

The idea of flocks [11] was presented as a framework for performing complex data analysis tasks on relational database systems. The method consists of a generator of candidate query parameter settings and their concomitant queries, and a filter which passes only those results that meet the specified condition. The canonical example for flocks is itemset counting. An earlier paper [8] has addressed the query optimization problems that arise when existing query optimizers are used in the context of flocks. Various means of pushing aggregate conditions down into the query execution plan were examined and a new logical operator *group-select* was defined to improve the flock execution optimization plan.

The emphasis of this paper is on the relationship between the system architecture and the flock execution plan. The choices involved are not independent: different optimization plans may result in the execution of the flock either internally, by optimizing the order of introduction of selection and grouping criteria on the underlying relations or externally, by means of auxiliary relations. Section 3 of this paper introduces the idea of auxiliary relations and their use in flocks execution. Section 4 elaborates on the generation of query flock plans. Section 5 reports on some experimental results applying these methods and choices to a large database of health-care data. In section 6 we conclude this paper.

2 Architecture

There are three different ways in which data mining systems use relational DBMS. They may not use a database at all, be loosely coupled, or be tightly coupled. We have chosen the tightly-coupled approach that does (almost) all of the data processing at the database. Before we justify our choice, we discuss the major advantages and drawback of the the other two approaches.

Most current data mining systems do not use a relational DBMS. Instead they provide their own memory and storage management. This approach has its advantages and disadvantages. The main advantage is the ability to fine-tune the memory management algorithms with respect to the specific data mining task.

Thus, the data mining systems can achieve optimal performance. The downside of this database-less approach is the lost opportunity to leverage the existing relational database technology developed in the last couple of decades. Indeed, conventional DBMS provide various extra features, apart from good memory management, that can greatly benefit the data mining process. For example, the recovery and logging mechanisms, provided by most DBMS, can make the results of long computations durable. Furthermore, concurrency control can allow many different users to utilize the same copy of the data and run data mining queries simultaneously.

Some data mining systems use a DBMS but only to store and retrieve the data. This loosely-coupled approach does not use the querying capability provided by the database which constitutes both its main advantage and disadvantage. Since the data processing is done by specialized algorithms their performance can be optimized. On the other hand, there is still the requirement for at least temporary storage of the data once it leaves the database. Therefore, this approach also does not use the full services offered by the DBMS.

The tightly-coupled approach, in contrast, takes full advantage of the database technology. The data are stored in the database and all query processing is done locally (at the database). The downside of this approach is the limitations of the current query optimizers. In was shown in [10] that performance suffers greatly if we leave the data mining queries entirely in the hands of the current query optimizers. Therefore, we need to perform some optimizations *before* we send the queries to the database, taking into account the capabilities of the current optimizers. To achieve this we introduce an *external optimizer* that sits on top of the DBMS. The external optimizer effectively breaks a complex data mining query into a sequence of smaller queries that can be executed efficiently at the database. This architecture is shown in Fig. 1.

The external optimizer can be a part of larger system for formulating data mining queries such as query flocks. The communication between this system and the database can be carried out in ODBC or JDBC.

3 Framework

3.1 Query Flocks

The query flock framework [11] generalizes the *a-priori trick* [1] for a larger class of problems. Informally, 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 to express the parameterized queries.
- A language to express the filter condition about the results of a query.

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

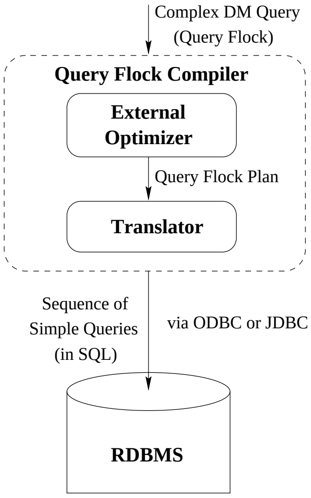

Fig. 1. Tightly-coupled integration of data mining and DBMS.

– One or more predicates that represent data stored as relations.
– A set of *parameters* whose names always begin with \$.
– A *query* expressed in the chosen query language, using the parameters as constants.
– A *filter 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 a query flock is a set of tuples that represent "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 checking whether the results pass the filter condition.

In this paper, we will describe query flocks using conjunctive queries [4], augmented with arithmetics, as a query language and an SQL-like notation for the filter condition. The following example illustrates the query flock idea and our notation:

Example 1. Consider a relation `diagnoses(Patient,Disease)` that contains information about patients at some hospital and the diseases with which they have been diagnosed. Each patient may have more than one disease. Suppose we are interested in pairs of diseases such that there are least 200 different patients diagnosed with this pair. We can write this question as a query flock in the following way:

QUERY:

```
answer(P) :- diagnoses(P,$D1) AND diagnoses(P,$D2)
             AND $D1 < $D2
```

FILTER:

```
COUNT(answer) >= 200
```

First, let us examine the query part. For a given pair of diseases ($D1 and $D2) the `answer` relation contains all patients diagnosed with the pair. The last predicate ($D1 < $D2) insures that the result contains only pairs of different diseases and does not contain a pair and its reverse. The filter part expresses the condition that there should be at least 200 such patients. Thus, any pair ($D1,$D2) that is in the result of the query flock must have at least 200 patients diagnosed with it.

Another way to think about the meaning of the query flock is the following. Suppose we have all possible disease values. Then, we substitute $D1 and $D2 with all possible pairs and check if the filter condition is satisfied for the query part. Only those values for which the result of the query part passes the test will be in the query flock result. Thus, the result of the query flock is a relation of pairs of diseases; an example relation is shown in Table 1.

Table 1. Example query flock result.

$D1	*$D2*
anaphylaxis	rhinitis
ankylosing spondylitis	osteoporosis
bi-polar disorder	insomnia

3.2 Auxiliary Relations

Auxiliary relations are a central concept in the query flock framework. An auxiliary relation is a relation over a subset of the parameters of a query flock and contains candidate values for the given subset of parameters. The main property of auxiliary relations is that all parameter values that satisfy the filter condition are contained in the auxiliary relations. In other words, any value that is not in an auxiliary relation is guaranteed not satisfy the filter condition. Throughout the examples in this paper we only consider filter conditions of the form $COUNT(ans) >= X$. However, our results and algorithms are valid for a larger class of filter conditions called *monotone* in [11] or *anti-monotone* in [9]. For this class of filter conditions, the auxiliary relations can be defined with subset of the goals in the query part of the query flock. For a concrete example, consider the query flock from Example 1:

Example 2. An auxiliary relation ok_d1 for parameter $D1 can be defined as the result of the following query flock:

QUERY:

```
ans(P) :- Diagnoses(P,$D1)
```

FILTER:

```
COUNT(ans) >= 200
```

The result of this flock consists of all diseases such that for each disease there are at least 200 patients diagnosed with it. Consider a pair of diseases such that there are at least 200 patients diagnosed with the pair. Then, both diseases must appear in the above result. Thus, the auxiliary relation is a superset of the result of the original query flock.

3.3 Query Flock Plans

Intuitively, a query flock plan represents the transformation of a complex query flock into a sequence of simpler steps. This sequence of simpler steps represent the way a query flock is executed at the underlying RDBMS. In principal, we can translate any query flock directly in SQL and then execute it at the RDBMS. However, due to limitations of the current query optimizers, such an implementation will be very slow and inefficient. Thus, using query flock plans we can effectively pre-optimize complex mining queries and then feed the sequence of smaller, simpler queries to the query optimizer at the DBMS.

A query flock plan is a (partially ordered) sequence of operations of the following 3 types:

Type 1 Materialization of an auxiliary relation
Type 2 Reduction of a base relation
Type 3 Computation of the final result

The last operation of any query flock plan is always of type 3 and is also the only one of type 3.

Materialization of an auxiliary relation: This type of operation is actually a query flock that is meant to be executed directly at the RDBMS. The query part of this query flock is formed by choosing a safe subquery [12] of the original query. The filter condition is the same as in the original query flock. Of course, there are many different ways to choose a safe subquery for a given subset of the parameters. We investigate several ways to choose safe subqueries according to some rule-based heuristics later in the paper. This type of operation is translated into an SQL query with aggregation and filter condition.

For an example of a step of type 1, recall the query flock that materializes an auxiliary relation for \$D1 from Example 2.

QUERY:

```
ans(P) :- Diagnoses(P,$D1)
```

FILTER:

```
COUNT(ans) >= 200
```

This materialization step can be translated directly into SQL as follows:

```
(1) ok_d1(Disease) AS
    SELECT Disease
    FROM diagnoses
    GROUP BY Disease
    HAVING COUNT(Patient) >= 200
```

Reduction of a base relation: This type of operation is a semijoin of a base relation with one or more previously materialized auxiliary relations. The result replaces the original base relation. In general, when a base relation is reduced we have a choice between several reducers. Later in this paper, we describes how to choose "good" reducers.

For an example of a step of type 2, consider the materialized auxiliary relation `ok_d1`. Using `ok_d1` we can reduce the base relation `diagnoses` as follows:

QUERY:

```
diagnoses_1(P) :- diagnoses(P,$D1) AND ok_d1($D1)
```

This can be translated directly into SQL as follows:

```
(2) diagnoses_1(Patient,Disease) AS
    SELECT b.Patient, b.Disease
    FROM diagnoses b, ok_d1 r
    WHERE b.Disease = r.Disease
```

Computation of the final result: The last step of every query flock plan is a computation of the final result. This step is essentially a query flock with a query part formed by the reduced base relations from the original query flock. The filter is the same as in the original query flock.

4 Algorithms

In this section we present algorithms that generate *efficient* query flock plans. Recall that in our tightly-coupled mining architecture these plans are meant to be translated in SQL and then executed directly at the underlying RDBMS. Thus, we call a query flock plan *efficient* if its execution at the RDBMS is efficient. There are two main approaches to evaluate the efficiency of a given query plan: cost-based and rule-based. A cost-based approach involves developing an appropriate cost model and methods for gathering and using statistics. In contrast, a rule-based approach relies on heuristics based on general principles, such as applying filter conditions as early as possible. In this paper, we focus on the rule-based approach to generating efficient query flock plans. The development of a cost-based approach is a topic of a future paper.

The presentation of our rule-based algorithms is organized as follows. First, we describe a general nondeterministic algorithm that can generate all possible query flock plans under the framework described in Section 3.3. The balance of this section is devoted to the development of appropriate heuristics, and the intuition behind them, that make the nondeterministic parts of the general algorithm deterministic. At the end of this section we discuss the limitations of conventional query optimizers and show how the query flock plans generated by our algorithm overcome these limitations.

4.1 General Nondeterministic Algorithm

The general nondeterministic algorithm can produce any query flock plan in our framework. Recall that a valid plan consists of a sequence of steps of types 1 and 2 followed by a final step of type 3. One can also think of the plan as being a sequence of two alternating phases: materialization of auxiliary relations and reduction of base relations. In the materialization phase we choose what auxiliary relations to materialize one by one. Then we move to the reduction phase or, if no new auxiliary relations have been materialized, to the computation of the final result. In the reduction phase we choose the base relations to reduce one by one and then go back to the materialization phase.

Before we described the nondeterministic algorithm in details we introduce the following two helper functions.

MaterializeAuxRel(Params, Definition) takes a subset of the parameters of the original query flock and a subset of the base relations. This subset forms the body of the safe subquery defining an auxiliary relation for the given parameters. The function assigns a unique name to the materialized auxiliary relation and produces a step of type 1.

ReduceBaseRel(BaseRel, Reducer) takes a base relation and a set of auxiliary relations. This set forms the reducer for the given base relation. The function assigns a unique name to the reduced base relation and produces a step of type 2.

We also assume the existence of functions *add* and *replace*, with their usual meanings, for sets and the function *append* for ordered sets. The nondeterministic algorithm is shown in Fig. 2

The number of query flock plans that this nondeterministic algorithm can generate is rather large. Infact, with no additional restrictions, the number of syntactically different query flock plans that can be produced by Algorithm 1 is infinite. Even if we restrict the algorithm to materializing only one auxiliary relation for a given subset of parameters, the number of query flock plans is more than double exponential in the size of the original query. Thus, we have to choose a subspace that will be tractable and also contains query flock plans that work well empirically. To do so effectively we need to answer several questions about the space of potential query flock plans. We have denoted these questions in Algorithm 1 with `(Q1) - (Q5)`.

(Q1) How to sequence the steps of type 1 and 2?
(Q2) What auxiliary relations to materialize?
(Q3) What definition to choose for a given auxiliary relation?

```
Algorithm 1
  Input: Query flock QF
         Parameters – set of parameters of QF
         Predicates – set of predicates in the body of the query part of QF
  Output: Query flock plan QFPlan
  // Initialization
    BaseRels = Predicates
    AuxRels = ∅
    QFPlan = ∅
  // Iterative Generation of Query Flock Plan
    while(true) do
(Q1)    choose NextStepType from {MATERIALIZE, REDUCE, FINAL}
        case NextStepType:
        MATERIALIZE: // Materialization of Auxiliary Relation
(Q2)      choose subset S of Parameters
(Q3)      choose subset D of BaseRels
          Step = MaterializeAuxRel(S, D)
          QFPlan.append(Step)
          AuxRels.add(Step.ResultRel)
        REDUCE: // Reduction of Base Relation
(Q4)      choose element B from BaseRels
(Q5)      choose subset R of AuxRels
          Step = ReduceBaseRel(B, R)
          QFPlan.append(Step)
          BaseRels.replace(B, Step.ResultRel)
        FINAL: // Computation of Final Result
          Step = MaterializeAuxRel(Parameters, BaseRels)
          QFPlan.append(Step)
          return QFPlan
        end case
    end while
```

Fig. 2. General nondeterministic algorithm.

(Q4) What base relations to reduce?
(Q5) What reducer to choose for a given base relation?

There are two main approaches to answering `(Q1) - (Q5)`. The first one involves using a cost model similar to the one used by the query optimizer within the RDBMS. The second approach is to use rule-based optimizations. As we noted earlier, in this paper we focus on the second approach.

In order to illustrate Algorithm 1, consider the following example query flock, that was first introduced in [11].

Example 3. Consider the following four relations from a medical database about patients and their symptoms, diagnoses, and treatments.

diagnoses(Patient, Disease) The patient is diagnosed with the disease.
exhibits(Patient, Symptom) The patient exhibits the symptom.
treatment(Patient, Medicine) The patient is treated with the medicine.
causes(Disease, Symptom) The disease causes the symptom.

We are interested in finding side effects of medicine, i.e., finding pairs of medicines `$M` and symptoms `$S` such that there are at least 20 patients taking the medicine and exhibiting the symptom but their diseases do not cause the symptoms. The question can be expressed as a query flock as follows:

QUERY:

```
ans(P) :- exhibits(P,$S) AND
          treatment(P,$M) AND
          diagnoses(P,D) AND
          NOT causes(D,$S)
```

FILTER:

```
COUNT(ans) >= 20
```

One possible query flock plan that can be generated by Algorithm 1 for the above query flock is shown in Table 2. This plan consists of a step of type 1 followed by two steps of type 2 and ending with the final step of type 3. The first step materializes an auxiliary relation `ok_s($S)` for parameter $S. The next two step reduce the base relations `causes(D,$S)` and `exhibits(P,$S)` by joining them with `ok_s($S)`. The last step computes the final result, relation `res($M,$S)`, using the reduced base relations.

Table 2. Example of a query flock plan produced by Algorithm 1.

Step	*Type*	*Result*	*QUERY*	*FILTER*
(1)	1	`ok_s($S)`	`ans_1(P) :- exhibits(P,$S)`	`COUNT(ans_1) >= 20`
(2)	2	`c_1(D,$S)`	`c_1(D,$S) :- causes(D,$S) AND ok_s($S)`	-
(3)	2	`e_1(P,$S)`	`e_1(P,$S) :- exhibits(P,$S) AND ok_s($S)`	-
(4)	3	`res($M,$S)`	`ans(P) :- e_1(P,$S) AND treatment(P,$M) AND diagnoses(P,D) AND NOT c_1(D,$S)`	`COUNT(ans) >= 20`

4.2 Levelwise Heuristic

First, we address the question how to sequence the steps of types 1 and 2 ((Q1)) along with the questions what auxiliary relations to materialize ((Q2)) and what base relations to reduce ((Q4)). The levelwise heuristic that we propose is loosely fashioned after the highly successful a-priori trick [1]. The idea is to materialize the auxiliary relations for all parameter subsets of size up to and including k in a levelwise manner reducing base relations after each level is materialized. So, starting at level 1, we materializing an auxiliary relations for every parameter. Then we reduce the base relations with the materialized auxiliary relations. At level 2, we materialize the auxiliary relations for all pairs of parameters, and so on. The general levelwise algorithm is formally described in Fig. 3.

```
Algorithm 2
   Input:  Query flock QF; K – max level
           Parameters – set of parameters of QF
           Predicates – set of predicates in the body of the query part of QF
   Output:  Query flock plan QFPlan
   // Initialization
      BaseRels = Predicates
      QFPlan = ∅
   // Levelwise Generation of Query Flock Plan
      for i = 1 to K do
         AuxRels_i = ∅
      // Materialization of Auxiliary Relations
         for all S ⊂ Parameters with | S |= i do
(Q3)        choose subset D of BaseRels
            Step = MaterializeAuxRel(S, D)
            QFPlan.append(Step)
            AuxRels_i.add(Step.ResultRel)
         end for
      // Reduction of Base Relations
         for all B ∈ BaseRels
(Q5)        choose subset R of AuxRels_i
            Step = ReduceBaseRel(B, R)
            QFPlan.append(Step)
            BaseRels.replace(B, Step.ResultRel)
         end for
      end for
   // Computation of Final Result
      Step = MaterializeAuxRel(Parameters, BaseRels)
      QFPlan.append(Step)
      return QFPlan
```

Fig. 3. General levelwise algorithm.

The levelwise heuristic has also some important implications on the choice of definitions of auxiliary relations and the choice of reducer for base relations discussed in the next two section.

4.3 Choosing Definitions of Auxiliary Relations

When choosing definitions of auxliary relations ((`Q3`)) there are two main approaches single and group. In the single approach, we choose a definition for a single auxiliary relation without regard to any other choices. In the group approach, in contrast, we choose definitions for several auxiliary relations at the same time. Thus, we can exploit existing symmetries among the parameters or equivalences among syntactically different definitions. Regardless of the particular approach we only consider definitions that form *minimal* safe subquesies, not involving a cartesian product. The subquesies are minimal in a sense that eliminating any subgoal will either make the subquery unsafe or will turn it into a cartesian product.

The already chosen levelwise heuristic dictates the use of the group approach in our algorithm. We can take advantage of the fact that we are choosing definitions for all auxiliary relations for a given level *simultaneously*. Thus, it is rather straightforward to use symmetries among parameters and equivalences among subqueries to choose the smallest the number of definitions that cover all auxliary relations. We refer to this strategy as the *least-cover* heuristic.

4.4 Choosing Reducers of Base Relations

When choosing a reducer for a given base relation we can employ two strategies. The first strategy is to semijoin it with the join of all auxliary relations that have parameters in common with the base relation. The second strategy is to semijoin it with all auxiliary relations that only have parameters appearing in the given base relation. With the second strategy we minimize the number of relations in the reduction joins while keeping the selectivity as high as possible. Again the use of the levelwise heuristic dictates our strategy choice. At the end of each level we have materialized auxiliary relations for all parameter subsets of the given size. Thus, the first strategy yields unnecessarily large reducers for every base relation at almost every level. Therefore, in our algorithm, we employ the second strategy.

4.5 K-Levelwise Deterministic Algorithm

Choosing the least-cover heuristic for (`Q3`) and the strategy outlined in Section 4.4 for (`Q5`) we finalize our algorithm that generates query flock plans. The formal description of the k-levelwise deterministic algorithm is shown in Fig.3.

```
Algorithm 3
  Input: Query flock QF; K – max level
         Parameters – set of parameters of QF
         Predicates – set of predicates in the body of the query part of QF
  Output:  Query flock plan QFPlan
  // Initialization
    BaseRels = Predicates; QFPlan = ∅
  // Levelwise Generation of Query Flock Plan, up to level K
    for i = 1 to K do
      AuxRels_i = ∅; MinDefs_i = ∅
    // find all minimal definitions of auxiliary relations
      for all S ⊂ Parameters with | S |= i do
        MinDefs_i.add(GetMinDefs(S, BaseRels))
      end for
    // choose least cover of minimal definitions
      Cover_i = GetLeastCover(MinDefs_i)
    // for each definition in the cover add corresponding
    // auxiliary realtions for all covered parameter subsets
      for all ⟨Def, CoveredParamSets⟩ ∈ Cover_i do
        for all S ∈ CoveredParamSets do
          Step = MaterializeAuxRel(S, Def)
          AuxRels_i.add(Step.ResultRel)
        end for
      // materialize the shared definition only once
        QFPlan.append(Step)
      end for
    // Reduction of Base Relations
      for all B ∈ BaseRels do
        R = ∅
      // choose reducer for base relation
        for all A ∈ AuxRels_i do
          if GetParams(A) ⊂ GetParams(B) then
            R.add(A)
        end for
        Step = ReduceBaseRel(B, R)
        QFPlan.append(Step)
        BaseRels.replace(B, Step.ResultRel)
      end for
    end for
  // Computation of Final Result
    Step = MaterializeAuxRel(Parameters, BaseRels)
    QFPlan.append(Step)
    return QFPlan
```

Fig. 4. K-Levelwise deterministic algorithm.

The k-levelwise deterministic algorithm uses the following three helper functions.

GetMinDefs(Params,Preds) takes a set of parameters and a set a of predicates (query). The function returns a tuple where the first element is the set of parameters and the second element is the set of all minimal definitions (subqueries) for the auxiliary relation for the given set of parameters.
GetLeastCover(Set of (Params,Defs)) takes a set of tuples composed of a set of parameters and a set of definitions. The function returns the smallest set of definitions that covers all sets of parameters using equivalences among syntactically different definitions.
GetParams(Pred) takes a predicate and returns the set of parameters that appear in the given predicate.

The query flock plan produced by Algorithm 3 with $k = 1$ for the query flock from Example 3 is shown in Table 3.

Table 3. Query flock plan produced by Algorithm 3 with $K = 1$.

Step	*Type*	*Result*	*QUERY*	*FILTER*
(1)	1	`ok_s($S)`	`ans_1(P) :- exhibits(P,$S)`	`COUNT(ans_1) >= 20`
(2)	1	`ok_m($M)`	`ans_2(P) :- treatment(P,$M)`	`COUNT(ans_2) >= 20`
(3)	2	`c_1(D,$S)`	`c_1(D,$S) :- causes(D,$S) AND ok_s($S)`	-
(4)	2	`e_1(P,$S)`	`e_1(P,$S) :- exhibits(P,$S) AND ok_s($S)`	-
(5)	2	`t_1(P,$M)`	`t_1(P,$M) :- treatment(P,$M) AND ok_m($M)`	-
(6)	3	`res($M,$S)`	`ans(P) :- e_1(P,$S) AND t_1(P,$M) AND diagnoses(P,D) AND NOT c_1(D,$S)`	`COUNT(ans) >= 20`

4.6 Comparison with Conventional Query Optimizers

Recall that we use query flock plans to insure the efficient execution of query flocks at the underlying RDBMS. The shortcomings, with respect to query flocks, of conventional query optimizers are the fixed shape (left-deep trees) of their query plans and the fact that aggregation is usually done last. Query flock plans rectify these problems by using reduction of base relations to circumvent the shape of the query plan and auxiliary relations to use aggregation on partial results as early as possible

The problem of including aggregation in query optimization is studied in [13, 6, 5]. In these papers, aggregation is pushed down, (or sometimes up), the query plan tree. The key difference with our work is that we use aggregation on a subset of the original query and the result is used to reduce the size of intermediate steps. Eventually the aggregation must be performed again but we have gained efficiency by having much smaller intermediate results.

5 Experiments

Our experiments are based on real-life health-care data. Below we describe a representative problem and the performance results.

Consider a relation `Diagnoses(PatientID,StayCode,Diagnose)` that contains the diagnoses information for patients during their stays at some hospital. Another relation, `Observe(PatientID,StayCode)`, contains the pairs of `PatientID` and `StayCode` for patients that are kept for observations for less than 24 hours. The rest of the patients are admitted to the hospital. Consider the following problem.

Find all pairs of diagnoses such that:

1. There are at least N patients diagnosed with the pair of diagnoses
2. At least one of them is an observation patient

We can express this problem naturally as a query flock:

QUERY:

```
ans(P,S) :- Diagnoses(P,S,$D1) AND
            Diagnoses(P,S,$D2) AND
            Diagnoses(Q,T,$D1) AND
            Diagnoses(Q,T,$D2) AND
            Observe(Q,T) AND
            $D1 < $D2
```

FILTER:

```
COUNT(ans) >= N
```

This problem is important to the hospital management because the reimbursement procedures and amounts for admitted and observation patients are different. Thus, management would like to identify some exceptions to the general trends, find their causes, and investigate them further for possible malpractice or fraud.

The `Diagnoses` relation contains more than 100,000 tuples, while the `Observe` relation contains about 8,000 tuples. We compared the performance of the 1-levelwise and 2-levelwise algorithms as well as the direct approach where the query flock is directly translated into SQL. We used a standard installation of ORACLE 8.0 running under Windows NT. The results are shown in Fig. 5.

For this dataset, the 2-levelwise algorithm outperforms the 1-levelwise algorithm more than 3 times. This result is somewhat surprising because the two

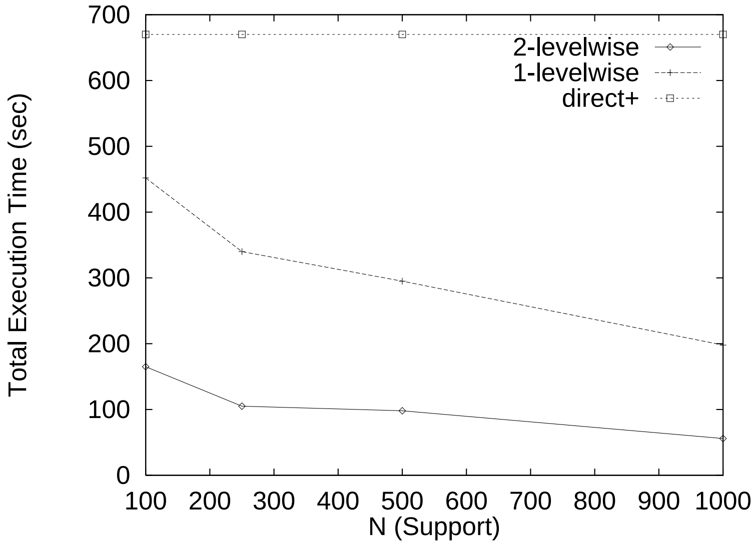

Fig. 5. Performance results on health-care data.

parameters \$D1 and \$D2 are symmetric (excluding the inequality) and thus, only one relation is materialized at level 1. However, the reduced base relation `Diagnoses` after level 1 is still rather large and the computation of the final result at this stage is much slower than materializing the auxiliary relation for the pair of parameters.

As expected, both algorithms perform much better than the direct approach where we translate the query flock directly in SQL. Infact, the actual translation did not finish executing in a reasonable amount of time. Thus, we had to augment the direct translation, hence *direct+*, with a preliminary step where we joined the `Observe` and `Diagnoses` relations. This step had the effect of reducing the size of the relations for two of the four `Diagnoses` predicates and eliminating the `Observe` predicate.

6 Conclusions

In this paper, we presented a tightly-coupled approach to integrating data mining and relational DBMS. We based our approach on the query flock framework where complex mining queries expressed as flocks are transformed into a query flock plan that consists of simpler queries. These queries can be optimized effectively by the query optimizer in the RDBMS. Thus, using query flock plans, we can execute complex mining queries efficiently in the RDBMS. We presented a class of levelwise algorithms for generating query flock plans. We also reported on some performance results that validate the effectiveness of our approach.

We are currently investigating cost-based optimization algorithms that interact with the internal optimizer of the RDBMS. Query flock plans produced by such algorithms could be even more efficient than the plans produced by rule-based algorithms. Our future work includes combining the rule-based and cost-based approaches to achieve optimal performance.

Acknowledgment: The authors would like to thank prof. Jeffrey Ullman for many discussions on the topic of query flocks.

References

1. R. Agrawal, T. Imilienski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 207–216, May 1993.
2. R. Agrawal and K. Shim. Developing tightly-coupled applications on ibm db2/cs relational database system: Methodology and experience. Research report, IBM Almaden Research Center.
3. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.
4. A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of 9th Annual ACM Symposium on the Theory of Computing*, pages 77–90.
5. S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 20st International Conference on Very Large Data Bases*, pages 354–366, Santiago, Chile, September 1994.
6. S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 167–182, Avignon, France, March 1996.
7. H. Houtsma and A. Swami. Set-oriented mining of association rules. In *Proceedings of International Conference on Data Engineering*, pages 25–33, Taipei, Taiwan, March 1995.
8. S. Nestorov and S. Tsur. Efficient implementation of query flocks. Technical report, Research and Development Lab, Hitachi America, Ltd., Santa Clara, California, September 1998.
9. R. Ng, L. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 13–24, Seattle, Washington, June 1998.
10. S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 343–354, Seattle, Washington, June 1998.
11. S. Tsur, J. Ullman, C. Clifton, S. Abiteboul, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: a generalization of association-rule mining. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 1–12, Seattle, Washington, June 1998.
12. J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I,II.* Computer Science Press, Rockville, Maryland, 1989.
13. W. Yan and P. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 345–357, Zurich, Switzerland, September 1995.