Knowledge Discovery from Client-Server Databases

Neil Dewhurst and Simon Lavington

Department of Computer Science, University of Essex, Wivenhoe Park, Colchester CO4 4SQ, UK
neil@essex.ac.uk, lavington@essex.ac.uk
WWW home page:
http://cswww.essex.ac.uk/Research/SystemsArchitecture/DataMining/Welcome.html

Abstract. The subject of this paper is the *implementation* of knowledge discovery in databases. Specifically, we assess the requirements for interfacing tools to client-server database systems in view of the architecture of those systems and of "knowledge discovery processes". We introduce the concept of a *query frontier* of an *exploratory process*, and propose a strategy based on optimizing the current query frontier rather than individual knowledge discovery algorithms. This approach has the advantage of enhanced genericity and interoperability. We demonstrate a small set of query primitives, and show how one example tool, the well-known decision tree induction algorithm C4.5, can be rewritten to function in this environment.

1 Introduction

Relational databases are the current dominant database technology in industry, and many organizations have collected large amounts of data in so-called *data warehouses* expressly for the purpose of decision support and data mining. In general the data must be queried in place: the question then arises of how knowledge discovery algorithms can be integrated with relational databases.

We will describe the architecture of a system being implemented at Essex University to address this problem. Essentially our proposal is that the client software should be in two halves, a "bottom half" which optimizes query transmission to the server, and a knowledge discovery algorithm which is implemented so as to give maximum scope for optimization. We believe it is possible to provide these services in a way which is to some extent *generic*, in the sense that it is independent of exactly which algorithm or tool is being implemented.

Most of the existing literature, including much of the work on scalable knowledge discovery algorithms [6], assumes that the database to be explored is stored in local main memory. But it is quite common to find a client accessing a server-resident database over a network, and the efficiency of algorithms as reported in published papers often depends on programming techniques that cannot be ported to this environment.

This is true, to take a particular example, of the well-known decision tree algorithm C4.5 [7], whose run-time efficiency is achieved by manipulating manipulating pointers into arrays of records. This technique cannot be applied to data stored in a relational database.

Moreover, knowledge discovery as it is practised in the real world is an iterative, multi-strategy process [1], and individually-optimized algorithms inevitably have limited interoperability. There is therefore a case for investigating methods of data access optimization whose benefits can be shared between multiple algorithms.

1.1 Framework: the concept of an "exploratory process"

We will abstract away from the details of the particular model or visualization tools being used, and assume only that there exists some *exploratory process* which acts as a source of queries and a consumer of result sets. The process has this general form: some set of initial queries is scheduled, after which the following two steps are repeated an unpredictable number of times:

- request a result set, optionally specifying that it should be one of a given subset of the outstanding queries
- schedule zero or more new queries

The intuition here is that the client can only examine one query result at a time, but can produce multiple queries simultaneously when an advance in the exploratory process opens up several new avenues at once. (Consider branching in a decision tree, or the task of filling in the probabilities in a Bayesian network.)

The result is that at any one time there is a set of outstanding queries, which we call the *query frontier*. Our approach to optimizing knowledge discovery is essentially to optimize the query frontier by combining individual queries into larger queries, a simple kind of multiple query optimization.

To exploit this kind of query optimization fully, knowledge discovery tools must be implemented so that queries are scheduled as soon as they become known. Later in this paper we describe our implementation of C4.5 as an example.

1.2 Client-server requirements

We assume that our principal aim is to minimise running times (as distinct from, for example, minimising network traffic). With that in mind, these are the requirements our system needs to satisfy.

Firstly, we need to ensure that work is divided between the server and client roughly according to their capacities, and that they are working simultaneously as much as possible. If either of these is not the case, there is an obvious waste of resources.

Secondly, since the server and network are both *shared* resources, they are subject to variable loading. Our query optimization should adjust dynamically

to increased load on the server by transferring more processing to the client, and to a busy network by fetching less data from the server (which probably implies doing more processing on the server).

2 Related work

The idea exploited in this paper, of optimizing database access by answering queries from the results of larger superqueries, is related to the materialization of selected views as described in [4]. For the view materialization problem, the set of queries to be optimized is known in advance, and the problem is to select among them: for knowledge discovery, the set of queries generally only emerges at runtime, so the superqueries must be generated dynamically.

The use of client-side caching to speed up processing for general database queries has a long history [8], [5]. Generally the idea is to cache query result sets on the client, indexed by some structure that supports *query matching* so it can be quickly determined whether a new query can be answered from the cache. The key concept is query *inclusion* - query q_1 includes query q_2 if q_2 can be answered by applying some further unspecified operation to q_1's result set.

A similar approach was used by [10] to implement a knowledge-base management system on top of an object-oriented database. They also address, as we do, the problem of balancing the work done by the server and client.

The BrAID system [9], which interfaces a logic-based AI system to a database, includes a subsystem that caches result sets and can reason with them.

3 Implementation of database queries

We summarize here some relevant background. A relational database query is expressed in SQL. Typically there is no client-side caching, so once the query has been parsed, it is processed entirely on the server. The philosophy is that there should be a simple division of labour between server and client: the server implements database operations, and the client implements a user interface. As [5] point out, this means that if the client is a reasonably powerful modern workstation, it tends to be under-utilised.

The parsed form of the SQL query is a tree of objects corresponding to physical database operations, called *iterators* [3]. An individual iterator might for example implement a table scan, a hash join, or a sort with consolidation. Each iterator is an object in the OOP sense of the word, implementing the same three methods: open(), fetch(), and close(). open() initialises the operation (this includes, but is not limited to, calling open on the iterator's children); fetch() retrieves the next row of the operation's result set; close() terminates the operation.

The time taken to perform a query may therefore be split into two phases, an *open phase* during which the client is idle, and a *fetch phase* during which rows are transferred from server to client, and both machines are working.

4 Primitive queries

Our technique for query optimization assumes that the queries emitted by the exploratory process are restricted in form. Specifically, we assume that the set of possible queries can be embedded in a larger set of queries which is a *semilattice* under the operation of query inclusion. Informally, this means that for any pair of queries q_1 and q_2, there is a unique "simplest" query Q which includes both q_1 and q_2. We denote Q by SUP(q_1, q_2). This property extends to any finite set of queries.

The larger set of queries, that is a semilattice, we refer to as the set of *primitives* of the exploratory process.

Our current implementation limits the primitives to one of three forms: the *select* primitive, the *count by group* primitive and the *ordered count by group* primitive. See Figure 1. (The select primitive, not shown, is a version of the count-by-group query without the GROUP BY clause.) These queries can be used as the basis for a wide range of knowledge discovery tasks [2]. They also provide good opportunities for splitting processing between client and server, because the count by group and ordered count by group primitives require either sorting or hashing large datasets using disks for temporary storage: both are expensive operations.

The ordered count-by-group primitive is used with databases that contain continuous (floating-point) data. Knowledge discovery tools typically handle continuous data by *discretizing* it, and the first step of a discretization algorithm is usually to sort the data. Our strategy is therefore to use the server to filter and sort the data, then use a client-side operator to calculate the discretized intervals as the sorted data streams off the network.

We will assume for the rest of this paper that the database contains only discrete values - in this case only the select and count-by-group primitives will be used.

From Figure 1 we can see that a count by group query has three parameters: a list of column names **a**, another list of (different) column names **b**, and a list of data values **v**. A typical query can therefore be denoted by CBG(**a**,**b**,**v**).

Definition 1. *The SUP of two count-by-group queries CBG(*$\mathbf{a},\mathbf{b},\mathbf{v}$*) and CBG(*$\mathbf{a}',\mathbf{b}',\mathbf{v}'$*) is CBG(*$\mathbf{a}^\star,\mathbf{b}^\star,\mathbf{v}^\star$*) where*
$\mathbf{b}^\star = [b_i : bi = b'i \wedge v_i = v'_i]$
$\mathbf{v}^\star = [v_i : b_i \in \mathbf{b}^\star]$
$\mathbf{a}^\star = \mathbf{a} \cup \mathbf{a}' \cup (\mathbf{b} - \mathbf{b}') \cup (\mathbf{b}' - \mathbf{b})$
The extension to select queries is straightforward. The SUP of two select queries is another select: the SUP of a select and a count-by-group is a count-by-group.

Example 1. Let A1, A2, A3 be attribute names.
The SUP of CBG([A1], [A2,A3], [2,3]) with CBG([A6], [A2,A3], [2,5])
is CBG([A1,A3,A6], [A2], [2]).

If q_1 and q_2 are count-by-group primitives and q_1 includes q_2, then q_2's result can be derived from q_1 by applying a combined projection, selection and aggregation operator, usually called *consolidation* in the OLAP literature.

```
SELECT a1, a2 .., am, COUNT(*)
FROM table
WHERE b1=v1 AND b2=v2 ...
GROUP BY a1, a2, .., am
COUNT BY GROUP
```

```
SELECT a*, a1, a2, .., am, COUNT(*)
FROM table
WHERE b1=v1 AND b2=v2 ...
GROUP BY a*, a1, a2, .., am
ORDER BY a*
ORDERED COUNT BY GROUP
```

Fig. 1. primitives for knowledge discovery

4.1 Optimization of the query frontier

The optimization of the query frontier is based on the following idea. Given two queries q_1 and q_2, let Q be SUP(q_1, q_2). We consider two possible joint query plans for this pair of queries: either we can execute both queries on the server, or we can execute Q=SUP(q_1, q_2) on the server and derive q_1 and q_2 from Q by consolidation on the client. We thus have a pair of competing *physical query plans*. It is easy to generalize this to the case where there is a set of n queries, $q_1 \cdot\cdot q_n$.
We consider two possible plans: to execute all the queries individually, or to derive each of the q_i from their SUP. The first plan has two open phases and (in general) a shorter fetch phase, compared to a single open phase plus a longer fetch. It is not obvious which of the plans will execute more quickly. We use a cost model, specific to a given client and server, to estimate for a given set of queries which of the two possible plans is the best.

4.2 Dynamic cost estimation

For cost estimation we use a pair of cost functions, denoted c_o and c_f, which estimate respectively the cost of the open phase and the fetch phase of the server query. We use two functions so the system can react flexibly to variable loading on the server, and varying network bandwidth. The estimated costs for a query plan are multiplied by a pair of factors reflecting the discrepancy between predicted and actual performance of previously executed query plans. At the start of a session both factors are set to one.

4.3 Implementation of the query frontier

The query frontier is optimized by partitioning it greedily and dynamically into disjoint subsets. At each turn one partition is selected, and its SUP is scheduled. Each subset of the frontier has a number associated with it, its *benefit*, which is an estimate of the advantage of scheduling that SUP query over scheduling its subqueries individually. (The benefit is calculated using the cost functions c_o and c_f.)

When the exploratory process schedules a new query q, the process is as follows. The query manager considers inserting q into each subset of the frontier,

and calculates the effect on that frontier's benefit: q is then inserted into the subset whose benefit it most improves. If it does not improve any of the subsets, it is inserted into a new subset on its own, with benefit zero.

4.4 Scheduling server queries

Each successive query's open phase is scheduled at the start of the preceding query's fetch phase, the aim being to ensure that both client and server are working continuously. See Figure 2 for an illustration.

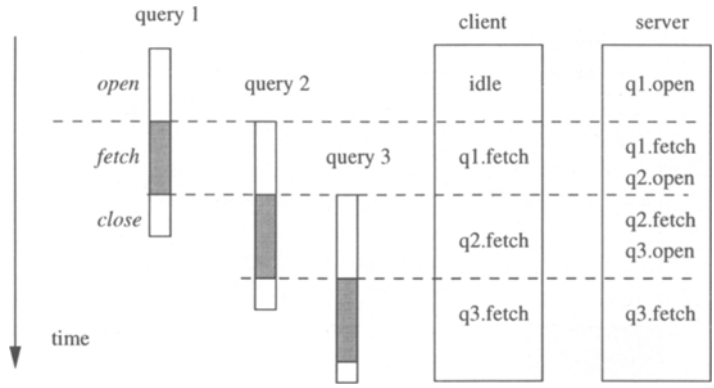

Fig. 2. query scheduling strategy

4.5 Client-side memory management

Client-side memory management is the responsibility of a separate software module, which manages both the main memory (where result sets are stored while they are created) and a persistent result-set cache on disk. When a dataset is cached, a new partition in the query frontier is created with a fixed root to represent it. The idea is that new queries from the exploratory process can be answered either by querying the server, or from cache if the data is available there. Client-only queries against the cache are implemented similarly to server queries, the difference being that rows are streamed off the disk rather than from the server.

5 Software architecture

The design of the query optimizer has some implications for the structure of the knowledge discovery tools that interface to it. Specifically, tools must be implemented using *callbacks*. The idea is that each query sent to the optimizer

has an ID attached to it, a number which uniquely identifies it to the application. When the result set is ready, it is passed to a result-set handler function, together with that ID. The initialization function initializes the model and schedules one

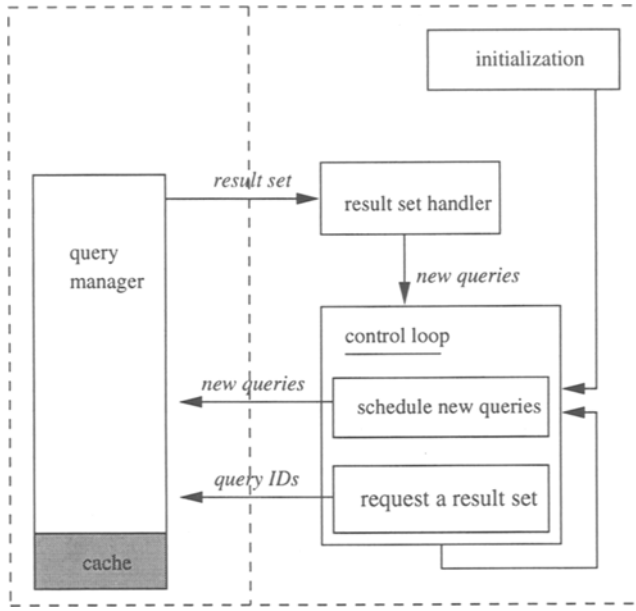

Fig. 3. skeleton "top half" architecture

or more initial queries. Control is then transferred to the control loop, which repeatedly requests a result set from the query manager, which replies to the result set handler. The result-set handler may pass one or more new queries to the control loop for scheduling.

The control loop should if possible avoid requesting a *specific* result set, to give the query manager maximum latitude for optimization. (Refer to the discussion of the query frontier in Section 1.1.) We will illustrate how this is done using the decision tree induction algorithm C4.5 [7].

5.1 Implementation example: C4.5

C4.5 as originally implemented builds the decision tree in strict depth-first order, and completes the tree before postpruning it. Our implementation removes both these constraints. The order in which the nodes are expanded depends on the order in which result sets are returned, which is left entirely to the query manager. Pruning of a node begins as soon as the whole subtree below the node has been finalized, and while other parts of the tree are still being built.

If (as we have assumed) the database contains only discrete data, C4.5 requires database access for three purposes:

- to calculate the "purity" of a node, in order to evaluate the stopping criterion (prepruning)
- to calculate the information gain for each candidate attribute at a given node (splitting)
- to calculate the revised error rate during postpruning (postpruning).

Each of these actions requires one or more count-by-group queries. Details of the calculations performed can be found in the Quinlan book [7]. We will use the following notation: PREPRUNE(N) denotes the query required for prepruning at node N; SPLIT(N,A) denotes the query needed to calculate the information gain of attribute A at node N; and POSTPRUNE(L) denotes the query needed at leaf node L during postpruning. The query ID that identifies a query encodes its type, a node number, and an optional attribute number.

In addition to these, we make use of a pseudoquery NOTIFY to allow a child node to "signal" its parent. This is implemented by calling the result-set handler recursively.

Each node of the decision tree is assigned a *state*. The node begins in state CALC-STOP, and proceeds through states CALC-SPLIT, WAIT, PRUNE-WAIT and several iterations of CALC-BRANCH, until it ends in state FINAL. The process terminates when the root node is in state FINAL.

The initialization step creates a single node (the root node, N_0) in state CALC-PRUNE, and schedules an initial query STOP(N_0). The action taken by the result-set handler when a result set is received for node N is then dependent on the state of N.

- **CALC-STOP**: The purity of the node is calculated, and the stopping criterion evaluated. If the criterion is satisfied, the node state is set to FINAL, and its parent node is notified.
- **CALC-SPLIT**: The incoming result set is for query SPLIT(N, A): the information gain for attribute A is calculated and stored. If all the attriibutes have been evaluated, the best split is chosen, a set of child nodes in state CALC-STOP is created, and STOP(N_i) is scheduled for each child N_i. The node state is set to WAIT.
- **WAIT**: The node remains in this state until a NOTIFY message has been received from each child. The node can then begin postpruning. One of the options considered is to replace the node with the root of its largest (in the sense of covering most cases) subtree, N_0 say. Each child in the tree below N_0 is moved to state CALC-BRANCH, and query POSTPRUNE(L) is scheduled for each leaf node L. The node state is set to PRUNE-WAIT.
- **PRUNE-WAIT**: The node receives a NOTIFY message from its largest child when the revised error estimates are done. Depending on the result, the node is either postpruned or not. Its parent is notified, and the node state is set to FINAL.
- **CALC-BRANCH**: If N is a leaf node, the incoming result set is used to calculate an estimated error rate for the node. Its parent is then sent a NOTIFY. A non-leaf node will decrement a count of its child nodes each

time a NOTIFY is received: when the count reaches zero the revised error rate for that node is calculated, and a NOTIFY sent to its parent.

6 Summary

We have described an architecture for the generic optimization of exploratory processes, based on the use of primitive queries which have a certain algebraic structure. Processing is split between client and server by the use of SUP queries, which are overlapped to ensure that client and server work in parallel. The problem of varying loads on the server and the network is addressed by using dynamically adjustable cost functions to choose between query plans. All the client-server issues identified at the outset have therefore been addressed in a way that allows efficient integration between knowledge discovery tools and database servers.

7 Acknowledgement

The research described in this paper was supported by the Engineering and Physical Sciences Research Council, grant number GR/L/16002.

References

1. Brachman, R.J., Anand,T: The Process of Knowledge Discovery in Databases: A Human-Centred Approach. In Usama M. Fayyad et al eds, Advances in Knowledge Discovery and Data Mining, AAAI Press (1996) 37–58
2. Freitas, A.A., Lavington, S.H.: Mining Very Large Databases with Parallel Processing. Kluwer Academic Publishers (1997)
3. Graefe, G. Query Evaluation Techniques for Large Databases. ACM Computing Surveys **25/2** (June 1993) 73–170
4. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing Data Cubes Efficiently. Proc. ACM SIGMOD Conference (1996) 205–216
5. Keller, A.M., Basu, J: A Predicate-Based Caching Scheme for Client-Server Database Architectures. The VLDB Journal **5** (1996) 35–47
6. Provost, F.J., Kolluri, V.: A Survey of Methods for Scaling Up Inductive Learning Algorithms. Proc. 3rd International Conference on Knowledge Discovery and Data Mining (1997)
7. Quinlan, J.R.: Programs for Machine Learning. Morgan Kaufman Publishers (1993)
8. Roussopoulos, N.: Materialized Views and Data Warehouses. Proc. 4th KRDB Workshop, Athens, Greece (1997) 12.1-12.6
9. Sheth, A.P., O'Hare, A.B.: The Architecture of BrAID: A System for Bridging AI/DB Systems Proc. 7th In. Conf. on Data Engineering (1991) 570–581
10. Thomas, J., Mitschang, B., Mattos, N., DeBloch, S.: Enhancing Knowledge Processing in Client/Server Environments. Proc. 2nd International Conference on Information and Knowledge Management (1993) 324–334