Estimating Query Result Sizes for Proxy Caching in Scientific Database Federations

Tanu Malik, Randal Burns*
Dept. of Computer Science
Johns Hopkins University
Baltimore, MD 21218

Nitesh V. Chawla†
Dept. of Computer Science and Engg.
University of Notre Dame
Notre Dame, IN 46556

Alex Szalay‡
Dept. of Physics and Astronomy
Johns Hopkins University
Baltimore, MD 21218

Abstract

In a proxy cache for federations of scientific databases it is important to estimate the size of a query before making a caching decision. With accurate estimates, near-optimal cache performance can be obtained. On the other extreme, inaccurate estimates can render the cache totally ineffective.

We present classification and regression over templates (CAROT), a general method for estimating query result sizes, which is suited to the resource-limited environment of proxy caches and the distributed nature of database federations. CAROT estimates query result sizes by learning the distribution of query results, not by examining or sampling data, but from observing workload. We have integrated CAROT into the proxy cache of the National Virtual Observatory (NVO) federation of astronomy databases. Experiments conducted in the NVO show that CAROT dramatically outperforms conventional estimation techniques and provides near-optimal cache performance.

Keywords: proxy caching, data mining, scientific federations

1 Introduction

The National Virtual Observatory (NVO) is a globally-distributed, multi-Terabyte federation of astronomical databases. It is used by astronomers world-wide to conduct data-intensive multi-spectral and temporal experiments and has lead to many new discoveries [Szalay et al. 2002]. At its present size – 16 sites – network bandwidth bounds performance and limits scalability. The federation is expected to grow to include 120 sites in 2007.

Proxy caching [Cao and Irani 1997; Amiri et al. 2003] can reduce the network bandwidth requirements of the NVO and, thus, is critical for achieving scale. In previous work, we demonstrated that bypass-yield (BY) caching [Malik et al. 2005] can reduce the bandwidth requirements of the Sloan Digital Sky Survey [SDSS] by a factor of five. SDSS is a principal site of the NVO. Proxy caching frameworks for scientific databases, such as bypass-yield caching, replicate database objects, such as columns (attributes), tables, or views, near clients so that queries to the database may be served locally, reducing network bandwidth requirements. BY caches load and evict the database objects based on their expected *yield*: the size of the query results against that object or, equivalently, the network savings realized from caching the object. The five-fold reduction in bandwidth is an upper bound, realized when the cache has perfect, *a priori* knowledge of query result sizes. In practice, a cache must estimate yield.

Similar challenges are faced by distributed applications which rely on accurate estimation of query result sizes. Other examples include load balancing [Poosala and Ioannidis 1996], replica maintenance [Olston et al. 2001; Olston and Widom 2002], grid computing [Serafini et al. 2001], Web caching [Amiri et al. 2003], and distributed query optimization [Ambite and Knoblock 1998]. In many such applications, estimation is largely ignored; estimation is an orthogonal issue and any accurate technique is assumed to be sufficient.

Existing techniques for yield estimation do not translate to proxy caching for scientific databases. Databases estimate yield by storing a small approximation of data distribution in an object. There are several obstacles to learning and maintaining such approximations in caches. First, proxy caches are situated closer to clients, implying that a cache learns distributions on remote data. Generating approximations incurs I/O at the databases and network traffic for the cache, reducing the benefit of caching. Caches and databases are often in different organizational domains and the requirements for autonomy and privacy in federations are quite stringent [Sheth and Larson 1990]. Second, a cache is a constrained resource in terms of storage. Thus, data structures for estimation must be compact. Finally, scientific queries

*e-mail: tmalik@cs.jhu.edu,randal@cs.jhu.edu
†e-mail:nchawla@cse.nd.edu
‡e-mail:szalay@jhu.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC2006 November 2006, Tampa, Florida, USA
0-7695-2700-0/06 $20.00 ©2006 IEEE

are complex and refer to many database attributes and join multiple tables. Traditional database methods typically perform poorly in such cases – they assume independence while building yield estimates from the component object data distributions. For such workloads, it is important to learn approximations of the joint data distribution of objects.

Several statistical techniques exist for learning approximate data distributions. These include sampling, histograms [Poosala and Ioannidis 1996], wavelets [Chakrabarti et al. 2000; Vitter and Wang 1999], and kernel density estimators [Gunopulos et al. 2000]. These techniques are considered in the context of query optimization within a database. Therefore, complete access to data is always assumed. Further, most of these techniques are not sufficiently compact when learning distributions for combinations of objects. For example, for histograms – the most popular method – the storage overhead and construction cost increase exponentially with the number of combinations [Vitter and Wang 1999].

We depart from current solutions and make yield estimates by learning the distribution of result sizes directly; we do not access data, nor do we attempt to estimate the underlying object distributions. A fundamental workload observation makes this possible. Queries in astronomy workloads follow *templates*, in which a template is formed by grouping query statements based on syntactic similarity. Queries within the same template have the same structure against the same set of attributes and relations; they differ in minor ways, using different constants and operators. Templates group workload into query families. The results size distributions of these families may be learned accurately by data mining the observed workload: queries in a template and the size of the results.

Our use of templates matches well with modern database applications in which forms and prepared statements are used to interrogate a database [Amiri et al. 2003; Luo and Xue 2004], resulting in queries within the same template. Scientific domains also include user-written queries. A user-written query is a scientific program which repeats a question on different databases or over different parts of a single database. This is remarkably exhibited by SDSS workload in which a month long workload of 1.4 million queries (both form-based and user-written) can be captured by 77 templates, and the top 15 templates capture 87% of all queries. Thus, it is sufficient for a BY cache to learn a template yield distribution, rather than learning distributions of thousands of objects plus their combinations.

We call our system for estimating yield classification and regression over templates (CAROT). From query statements and results within a template, CAROT learns a yield distribution through the machine learning techniques of classification trees and regression. Classification trees learn the distribution crudely and regression refines it. Future queries to the same template are estimated from a learned yield distribution and are also used to re-evaluate the yield distribution. If required, they tune the learned model. The computational complexity of both classification trees and regression is very low, allowing CAROT to generate and update learning models quickly and asynchronously. The sizes of learned models in CAROT vary from tens to hundreds of kilobytes allowing them to fit entirely in memory, making CAROT extremely space efficient. Through its use of templates, CAROT captures queries that refer to multiple objects, enabling CAROT to estimate complex queries accurately, *e.g.* multi-dimensional range queries and user-defined functions.

Extensive experimental results on the BY caching framework show that CAROT outperforms other techniques significantly in terms of accuracy and meets the requirements on compactness and data-access independence. On the SDSS workload, BY caching plus CAROT achieves savings within 5% of optimal, as compared to 45% with the Microsoft SQL optimizer, which uses histograms and refinement [Ioannidis 2003]. We include: an analysis of the SDSS workload, results that show CAROT to be compact and fast, and a detailed comparison of CAROT and the optimizer on prevalent queries.

2 Related Work

The performance of BY caches depends critically upon accurate query result size estimates. This is also true for query optimizers. Query optimizers choose the most efficient query execution plan based on cost estimates of database operators. Query optimizers rely on statistics obtained from the underlying database system to compute these estimates. In this section, we review the prominent statistical methods used by query optimizers for cost estimation and consider their applicability within BY caches.

Previous work for estimation in query optimizers can be classified in four categories [Chen and Roussopoulos 1994], namely non-parametric, parametric, sampling and curve-fitting. We first consider data dependent methods within each category. Non-parametric approaches [Poosala et al. 1996; Gunopulos et al. 2000; Ioannidis 2003] are mainly histogram based. Histograms approximate the frequency distribution of an object by grouping object values into "buckets" and approximate values and frequencies in each bucket. Parametric methods [Selinger et al. 1979; Christodoulakis 1983] assume that object data is distributed according to some known functions such as uniform, normal, Poisson, Zipf, etc. These methods are accurate only if the actual data also follows one of the distributions. Sampling based approaches [Lipton and Naughton 1995] randomly sample object data to obtain accurate estimates. Curve-fitting approaches [W. Sun and Deng 1993; Chen and Roussopoulos 1994] assume data distribu-

tion to be a polynomial that is a linear combination of model functions and use regression to determine the coefficients of the polynomial.

All the above approaches require access to data. Recently, curve-fitting methods and histograms have been improved to include query feedback, *i.e.,* the result of a query after being executed.

Chen and Roussopoulos [Chen and Roussopoulos 1994] first suggested the idea of using query feedback to incrementally refine coefficients of a curve fitting approach. Their approach is useful when distributions are learned on single objects and the data follows some smooth distribution. Neither of these assumptions are valid for BY caches in which cache requests refer to multiple objects and arbitrary data sets have to be approximated. In the same context, Harangsri *et al* [Harangsri et al. 1997] used regression trees, instead of a curve fitting approach. Their approach is also limited to single objects. Further, it remains an open problem if one can find model functions for a combination of objects [Aboulnaga and Chaudhuri 1999].

Aboulnaga and Chaudhari [Aboulnaga and Chaudhuri 1999] use the query feedback approach to build self-tuning histograms. The attribute ranges in predicate clauses of a query and the corresponding result sizes are used to select buckets and refine frequency of histograms initialized using uniformity assumption. Both single and multi-dimensional (m-d) histograms can be constructed by this method. Higher accuracy on m-d histograms is obtained by initializing them from accurate single dimensional histograms. However the latter can only be obtained by scanning data, making such approaches data dependent. The other approach of initializing m-d histograms with uniformly-distributed, single-dimensional histograms involves expensive restructuring and converges slowly. Further, the method is limited to range queries and does not generalize when the attributes are, for example, user-defined functions. STHoles [Bruno et al. 2001] improves upon self-tuning histograms, but is subject to the same limitations. Workload information has also been incorporated by LEO [Stillger et al. 2001] and SIT [Bruno and Chaudhuri 2002] in order to refine inaccurate estimates of the query optimizers. These approaches are closely tied to the query optimizer and, thus, are not applicable to the caching environment.

CXHist [Lim et al. 2005] builds workload-aware histograms for selectivity estimation on a broad class of XML string-based queries. XML queries are summarized into feature distributions, and their selectivity quantized into buckets. Finally, they use naive Bayes classifiers to compute the bucket to which a query belongs. The naive Bayes approach assumes conditional independence among the features within a bucket. Conditional independence does not hold in scientific workloads, such as the SDSS. Astronomy consists of highly correlated spatial data. Assuming independence among spatial attributes leads to incorrect estimates in regions without data.

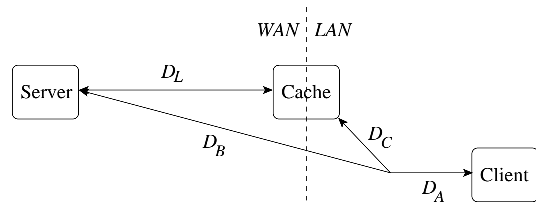


Figure 1: Network Flows in a Client-Server Pair Using Bypass-Yield Caching

In CAROT, we use a curve-fitting approach. But instead, of approximating the frequency distribution of the data, we approximate the yield function in the parameter space, in which parameters are obtained from query statements. For the specific case of multi-dimensional range queries, this reduces to approximating the cumulative frequency distribution as a piece-wise linear function, in which the pieces are decided from decision trees and regression learns the linear function.

3 Yield Estimation in BY Caches

In this section, we describe the architecture of the bypass-yield cache, the use of query result size estimates in cache replacement algorithms, and metrics which define the network performance of the system. Then, we highlight properties of BY caches that make yield estimation challenging.

3.1 The BY Cache Application

The bypass-yield cache is a proxy cache, situated close to clients. For each user query, the BY cache evaluates whether to service the query locally, loading data into the cache, versus shipping each query to be evaluated at the database server. The latter is a *bypass* query, because the query and its results circumvent the cache and do not change its contents. By electing to bypass some queries, caches minimize the total network cost of servicing all the queries. Figure 1 shows the architecture in which the network traffic minimized (WAN) is the bypass flow from a server directly to a client D_B, plus the traffic to load objects into the cache D_L. The client application sees the same query result data D_A for all caching configurations, *i.e.,* $D_A = D_B + D_C$, in which D_C is the traffic from the queries served out of the local cache.

In order to minimize the WAN traffic, a BY cache management algorithm makes an economic decision analogous to the rent-or-buy problem [Fujiwara and Iwama 2002]. Algorithms choose between loading (buying) an object and servicing queries for that object in the cache versus bypassing the cache (renting) and sending queries to be evaluated at

sites in the federation. (Recall that objects in the cache are tables, columns or views.) Since objects are much larger than query results or, equivalently, buying is far more costly than renting, objects are loaded into the cache *only* when the cache envisions long-term savings. At the heart of a cache algorithm is byte yield hit rate (BYHR), a savings rate, which helps the cache make the load versus bypass decision. BYHR is maintained on all objects in the system regardless of whether they are in cache or not. BYHR is defined as

$$\text{BYHR} \equiv \sum_j \frac{p_{i,j} y_{i,j} f_i}{{s_i}^2} \qquad (1)$$

for an object o_i of size s_i and fetch cost f_i accessed by queries Q_i with each query $q_{i,j} \in Q_i$ occurring with probability $p_{i,j}$ and yielding $y_{i,j}$ bytes. Intuitively, BYHR prefers objects for which the workload yields more bytes per unit of cache space. BYHR measures the utility of caching an object (table, column, or view) and measures the rate of network savings that would be realized from caching that object. BY cache performance depends on accurate BYHR, which, in turn, requires accurate yield estimates of incoming queries. (The per object yield $y_{i,j}$ is a function of the yield of the incoming query.)

3.2 Challenges

BY caching presents a challenging environment for the implementation of a yield estimation technique. An estimation technique, even if accurate, can adversely affect performance in terms of both resource usage and minimizing overall network traffic. We describe below some of the main challenges.

Limited Space: Caches provide a limited amount of space for metadata. Overly large metadata has direct consequences on the efficiency of a cache [Drewry et al. 1997] and consumes storage space that could otherwise be used to hold cached data. Therefore, metadata for yield estimation must be concise.

Remote Data: The BY cache acts as a proxy, placed near the clients and far from servers. This automatically creates an access barrier between the cache and the server. Management boundaries create privacy concerns making it difficult to access the databases to collect statistics on data. The data is mostly accessed via a Web services interface. Further, a cache may choose not to invest its resources in the I/O-intensive process of collecting statistics on data [Chen and Roussopoulos 1994].

Variety of Queries: BY caching serves scientific workloads which consist of complex queries. A typical query contains multiple clauses, specifying multi-dimensional ranges, multiple joins, and user-defined functions. A yield estimation technique should be general. Specifically, it should not assume independence between clauses to estimate the yield.

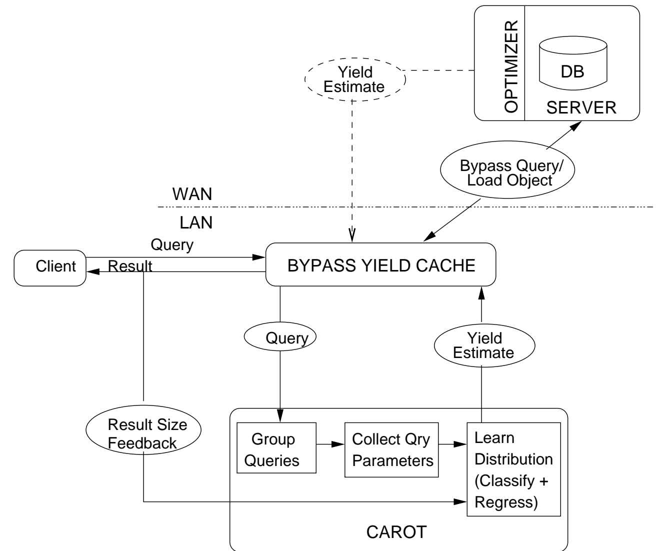


Figure 2: Estimating Yield in a BY Cache

4 CAROT

We describe CAROT (Section 4.1), the yield estimation technique used by the BY cache for most queries (Figure 2). CAROT groups similar queries together and then learns the yield distribution in each group. To learn the distribution, CAROT uses query parameters and query results as inputs, *i.e.,* it does not collect samples from data to learn the distribution. CAROT uses the machine learning techniques of classification and regression, which result in concise and accurate models of the yield distribution. The yield model in CAROT is self-learning. When a query arrives, CAROT estimates its yield using the model. When the query is executed, the query parameters and its result together update the model. This results in feedback-based, incremental modeling of the yield distribution. In Section 4.2 we illustrate the learning process in CAROT using a real-world example from Astronomy.

In order to provide accurate estimates for all queries, CAROT occasionally prefers an estimate from the database optimizer in favor of its own estimate (Section 4.3). This happens for (1) queries in templates for which enough training data has not been collected to learn a distribution, and (2) queries in which the database has specific information that gives it advantage, *e.g.,* uniqueness or indexes.

4.1 Estimating Query Yield from CAROT

Grouping Queries: CAROT groups queries into *templates*. A template τ over SPJ (Select-Project-Join) queries and user-defined functions (UDFs) is defined by: (a) the set of objects mentioned in the `from` clause of the query (objects imply tables, views or tabular UDFs) and (b) the set of attributes and

```
Select #
From R
Where R.a ?1 @1
and   R.b ?2 @2

# = {TOP,MAX}
? = {<,>,=,!=}
@ = {constants}
```

(a) Template and its parameters

```
Parameter Space Yield
{#,?1,@1,?2,@2,    Y}
<----------------> <-->
{1,<,50,>,0.1,     1}
{0,>,23,<,0.6,   250}
{0,<,50,<,0.9,  2000}
{0,<,68,>,0.1,   790}
{0,<,159,>,0.3, 3890}
```

(b) Yield distribution

```
Yield      CF
 <=100     A
 101-1000  B
 >=1000    C

 {(p1,A),(p2,B),
  (p3,C),p4,B),
  (p5,C)}
```

(c) Class distribution

Figure 3: Learning in CAROT

UDFs occurring in the predicate expressions of the `where` clause. Intuitively, a template is like a function prototype. Queries belonging to the same template differ only in their parameters. For a given template τ, the parameters (Figure 3(a)) are: (a) constants in the predicate expressions, (b) operators used in the join criteria or the predicates, (c) arguments in table valued UDFs or functions in predicate clauses, and (d) bits which specify if the query uses an aggregate function in the select clause. A template does not define the choice of attributes in the select clause of a user query and the parameters of a template are those features in a query that influence/determine its yield.

Collecting Query Parameters: The parameters of a query that influence its yield form a vector. Figure 3(b) shows vectors with parameters and yields from a few queries. For a template, the parameter space refers to all the possible combination of parameters, each chosen from their respective domain. A query belonging to a template refers to one of these combinations and has a corresponding yield in the range $\mathscr{R} = (0, max(t_i))$, in which $max(t_i)$ is the maximum yield of a query in a given template, *i.e.,* the size of a relation for a query to a single relation and the size of the cross product for a join query.

CAROT learns an approximate distribution of yields within each template. The actual yield distribution of a template τ over n queries is the set of pairs $\tau_D = (p_1, y_1), (p_2, y_2), \ldots, (p_n, y_n)$, in which p_i is the parameter vector of query q_i, and y_i is its corresponding yield. CAROT groups queries within a template into classes and learns an approximate, class distribution, defined as $\tau_C = (p_1, c_1), (p_2, c_2), \ldots, (p_n, c_n)$ in which $c_i = C_F(y_i)$ and C_F is a classification function.

Classification and Regression: Having received n queries in a template, CAROT learns the class distribution using decision trees [Quinlan 1993]. Decision trees recursively partition the parameter space into axis orthogonal partitions until there is exactly one class (or majority of exactly one class) in each partition. They do it based on information gain of parameters so as to minimize the depth of recursion. CAROT uses decision trees as they are a natural mechanism for learning a class distribution in the parameter space when independence among parameter values cannot be assumed.

By learning τ_C, *i.e. classes* of yields, and not τ_D, *i.e. values* of yields, CAROT loses some information. It regains some of the lost information by constructing a linear regression function within each class. A class specific regression function gives yield values for different queries that belong to the same class.

Finally, CAROT uses k-means clustering as the classification function C_F in which k is the number of classes and is a dynamically, tunable parameter. Several techniques [Pelleg and Moore 2000; Hamerly and Elkan 2003] can be used as a wrapper over k-means to find a suitable k from the lowest and highest observed yield value, or it can be chosen based on domain knowledge.

Refinement: Once the cache/server has used the estimate and served the query result, the size of the result is used as a feedback for CAROT. This feedback, combined with parameters of the query, refine the learned class distribution. When updating the decision tree, CAROT also updates the corresponding linear regression function(s). We update the learning model after collecting the fixed n number of queries in a template.

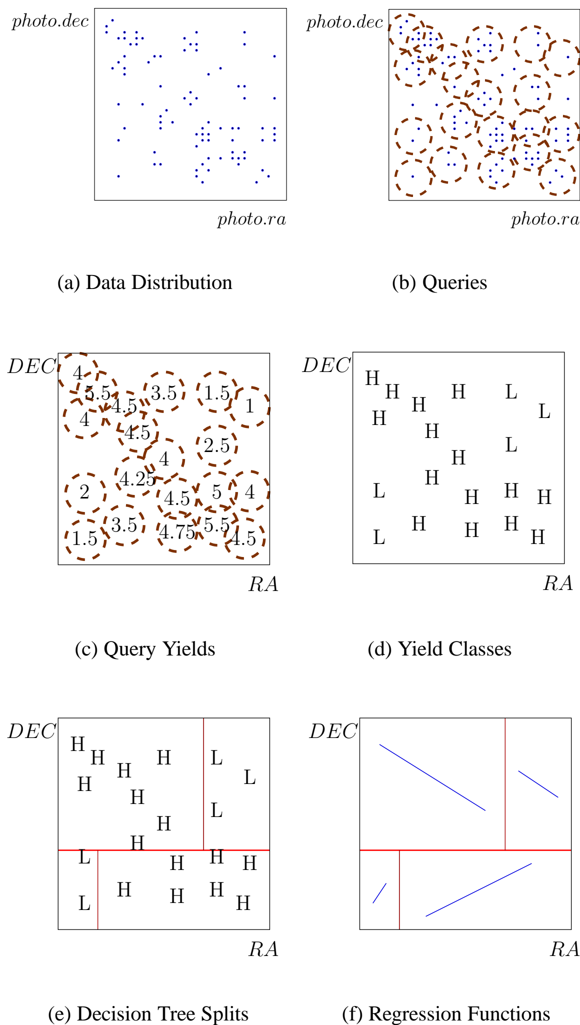
4.2 A Multi-Dimensional Example

We illustrate CAROT using an example astronomy template from the SDSS workload. In particular, we show how CAROT accurately estimates the distribution of data in a combination of attributes using only queries and their results. Figure 4(a) depicts the data distribution in a relation called *photo* that stores spatial location of astronomical bodies. The attributes *photo.ra*, *photo.dec* are spherical coordinates used by astronomers to locate an astronomical body. Each point in the figure corresponds to an astronomical body and has a spatial location given by the *ra* and *dec* attributes of the table. The figure shows relatively higher density of astronomical bodies in the top-left and bottom-right corners. Thus, *ra* and *dec* are not independent in this distribution. In other words, the joint probability is not the product of the individual probabilities of *ra* and *dec*. It is not possible to represent this distribution using a combination of single dimensional histograms on *ra* and *dec*. Thus, a query optimizer cannot estimate yield as accurately as CAROT for queries from this distribution.

Now we describe how CAROT learns this distribution *indirectly*; it learns it from queries in the workload, not data. One of the popular queries in astronomy involves the user-defined function, *getNearbyObject*. The function is often used as:

```
SELECT * FROM Photo p,
getNearbyObject(@ra,@dec) g
WHERE p.objid = g.objid
```

in which `@ra` and `@dec` are parameters supplied by the user. In reality, the function takes a radius parameter as well, but



(a) Data Distribution (b) Queries

(c) Query Yields (d) Yield Classes

(e) Decision Tree Splits (f) Regression Functions

Figure 4: CAROT on SDSS Data

we assume it to be constant for simplicity of showing this example in 2-D. Figure 4(b) shows queries matching this template on the data distribution. Each circle shows the neighborhood from which the objects are fetched. Figure 4(c) shows the same queries in the parameter space as seen at the BY cache, which has no knowledge of the distribution. Each query also has a corresponding yield – shown as the logarithm of the query result size in bytes – and is a function of the parameter `@ra` and `@dec`.

The high variance in yield for queries in this template gives CAROT a yield distribution to learn. It learns by observing that some queries are high (H) in yield (logarithmic value greater than 3.5) and some low (L) (Figure 4(d)). It classifies this distribution using decision trees, which split on specific values of *RA* and *DEC*, learning regions of *purely* high density and *purely* low density. Figure 4(e) shows an initial split on a *DEC* and then a split on *RA*. The decision tree itself can be used for estimation, but we improve upon it by using regression. In each decision tree leaf node, we look at the original yield values for queries and approximate this distribution through linear regression. We show this pictorially in Figure 4(f). We show straight lines, although the regression function is a plane.

4.3 Falling Back on the Optimizer

To provide a consistent yield estimation service to the BY cache, CAROT identifies circumstances in which it cannot provide accurate estimates and chooses the optimizers estimates in favor of its own.

This arises in two situations. (1) New queries that cannot be grouped into existing templates are estimated at the optimizer. CAROT learns the distribution after a few queries to a new template have occurred. Further queries are estimated from the learned distribution, after which learning and estimation go hand-in-hand. As experiments show, such cases are few, given the few number of templates and the rapid rate of learning due to locality in queries. (2) There are queries that are better estimated at the optimizer. These include queries for which there are indexes or uniqueness constraints. While CAROT can learn such distributions, the optimizer has specific information on which to make more accurate estimates. In many such cases, the optimizer produces exact answers and CAROT would expend space and time to learn a trivial distribution, *e.g.* uniqueness. In both cases, CAROT adopts a simple policy, it detects such queries from template and schema information and returns control to the BY cache, which obtains estimates from the database optimizer.

Were an optimizer not available, CAROT estimates such queries heuristically. For example, identity queries against a key field always return one row. This will not be the case for the SDSS site in the NVO, for which we invoke the MS SQL optimizer. However, some federations may not expose the database optimizer interface.

5 Implementing CAROT

We describe how CAROT identifies templates for a large class of queries and extracts feature vectors. While our treatment of detecting templates and extracting feature vectors is general, we highlight the cases that dominate the SDSS workload.

5.1 Template Detection

Queries submitted by applications often use or follow a template. In a template, queries are constructed from a *prepared*

statement and parameters to queries are submitted by user programs, *i.e.*, each query is an instantiation of a particular template. We previously argued that Web forms and application derived queries make templates prevalent.

In CAROT, the template generator has two goals: (a) infer templates of incoming queries from a database of templates and (b) detect new query templates from changing user queries. To do so, we convert predicate expressions into a canonical form in order to detect similarity, even when the queries do not take the exact same form; *e.g.*, the order of two commutative terms in a predicate expression are reversed. To construct the canonical form of a query, we first rewrite the expression in AND-OR normal form and order the clauses according to a numerical labeling of the database schema. For example, attribute 2.1 is the first attribute in the second table.

Based on the canonical form of the predicate expression, we construct a template *signature* used to match queries to existing templates. A signature is a unique, numeric value derived from the attributes and operators used in the predicate expression. This allows for fast template matching. Template detection builds upon existing query matching algorithms [Amiri et al. 2003; Luo and Naughton 2001; Luo and Xue 2004].

5.2 Constructing Parameter Vectors

It is important that parameter vectors encode the maximum information available in query parameters. Thus, we include both syntactic and semantic information by transforming the input parameters. We describe some of the important transformations on the most common predicates as witnessed in the SDSS workload.

- Predicate clauses of the form "`col i_op constant`" in which `i_op` is a relational operator from $<, >, =, >=, <=$ and `col` is an attribute, are represented by the feature vector `(constant,i_op)`.
- A predicate clause of form "`col BETWEEN constant1 and constant2`" is represented as `(constant1,constant2-constant1)`. The difference captures the distance between the two constants and has more semantic information than other feature vectors, such as `(constant1,constant2)`.
- For user-defined functions, whether in table or predicate clause, we use the arguments to the function as a feature vector, *i.e.* `(arg1,arg2,...,argn)`. The arguments can further be transformed by including semantic information from user-defined functions.
- Aggregate functions such as COUNT, MAX, MIN and constructs such as TOP in select clause may change yield drastically. We associate a single feature to indicate the presence of these primitives in a query.

Number of queries	1,403,833
Number of query templates	77
Percent of queries in top 15 templates	87%
Yield from all queries	1706 GB
Percent yield in top 35 templates	90.2%

Table 1: SDSS Astronomy Workload Properties

The concept of `col`, *i.e.,* a "column", is general and defined recursively as compositions of (a) a single attribute of a relation, (b) a scalar user-defined function, which returns a numeric value, and (c) a bit operator (`b_op` $\in (|, \&)$) expression of the form `col1 b_op col2` or `col1 b_op constant`. For a given template, when arguments or constants change in columns, we consider them as predicates and vectors are formed as defined previously.

Our system parses queries with numeric constants, operators and strings. It does not, at this point, generalize to string operators, such as `LIKE` or `IN`, which may take arbitrary strings.

6 Experimental Results

CAROT is a system developed for BY caches. BY caches are being currently introduced in the mediation middle-ware for the NVO federation of astronomy databases [FedCache]. From the current 16 databases of the federation, we considered the SDSS database and its corresponding workload to evaluate CAROT.

We present two levels of experiments. At the macro level we show performance of CAROT when used along with BY caches. In particular, we show the effect of yield estimates on the cache's network performance: the most important cost metric to measure the efficacy of the BY cache. CAROT outperforms the Microsoft SQL optimizer and preserves the network-savings realized by BY caching when compared with having exact, prior knowledge of query yield. Subsequently, we conduct a fine-grained evaluation of the tool on the most important queries in the workload. These experiments explore the accuracy of CAROT's estimates, the size of its data structure, and the running time of learning algorithms.

6.1 Experimental Setup

Workload Characterization: We took a month-long trace of queries against Sloan Digital Sky Survey (SDSS) database. The SDSS is a major site in the National Virtual Observatory: a federation of astronomy databases. This system is actively used by a large community of astronomers. Our trace has more than 1.4 million queries that generate close to two Terabytes of network traffic.

Template	Number of Queries	Dimensions in Feature Vector	Available Index	Query Semantics
T1	23343	4	1	Range query on a single table
T2	103148	6	0	Function query on a single table
T3	761605	4	0	Function query on a single table
T4	4142	3	0	Function query on 2 joined tables
T5	68603	4	1	Equality query on a single table
T6	3176	5	0	Function query on 3 joined tables

Table 2: Six Important Query Templates in the SDSS Workload.

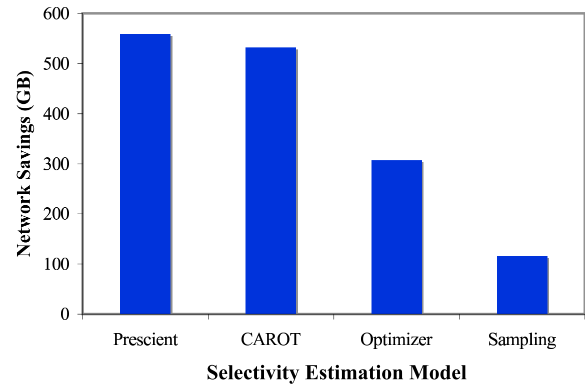
An analysis of the traces reveal that an astonishingly small number of query templates capture the workload (Table 1). A total of 77 different query templates occur in all 1.4 million queries. Further, the top 15 of these templates account for 87% of the queries, and the top 35 account for 90% of the total network traffic (yield). This confirms our intuition that most queries are generated through either Web forms and application libraries or correspond to user-written queries in iterative scientific programs. Thus, the SDSS traces exhibit a remarkable amount of reuse. The trace we considered included a variety of access patterns, such as range queries, spatial searches, identity queries, and aggregate queries.

CAROT's scalability and accuracy are derived from workload properties. Storage requirements scale with the number of templates. Because few templates capture the SDSS workload, CAROT has minimal space requirements. Also, few templates mean that a large number of queries appear in each template. This gives CAROT good training data, based on which it learns query yields accurately.

The SDSS Database: In SDSS, we used the 2.0TB BESTDR4 database. The database has 120 tables, and 5948 columns in total. The largest table has 446 columns. BESTDR4 database is stored in Microsoft SQL 2000 server.

The BY Cache: We were provided with a prototype of the BY cache. The prototype implements the OnlineBY cache algorithm. It maintains a virtual cache, the total size of which is equal to 30% of the database size, which in our case is the BESTDR4 database. A 30% cache size equals 600GB, which is just large enough to accommodate the largest table of BESTDR4. 0.5% of the cache space is kept for metadata (schema information) and internal data structures. BY caches cache database objects, and need a minimum cache size equal to the size of the largest object.

Comparison Methods: To compare CAROT, we use two other selectivity estimation methods: **Sampling** and **Optimizer**. In **Sampling**, we randomly sampled 1GB of data from the SDSS database. Choosing a 1GB large sample meant the cache allocated 1GB of its space to the random sample. Given the space constraints at the cache, this is the maximum we think a cache should allocate. Also given CAROT's small storage requirements, 1GB of space is substantial. For the **Optimizer** method, we use the Microsoft SQL2000 query optimizer. In this method, query estimates



(a) Network Cost

Model	Network Cost (GB)	Savings (GB)	% difference from optimal
No Caching	1706.28		
Prescient	1147.51	558.49	-0.00
CAROT	1174.13	532.15	-4.72
Optimizer	1399.06	307.22	-45.01
Sampling	1590.81	115.47	-79.32

(b) Savings comparison

Figure 5: Impact of Yield Estimation on Bypass-Yield Caching Performance

are obtained from the optimizer. There were two reasons for choosing the optimizer. First, the optimizer method occupies no space in the cache as the estimates are stored at the server. Thus space-wise this may be the best method for BY caches. Second, optimizer estimates in MS SQL 2000 are stored on a per object basis as max-diff histograms, which allows us to compare CAROT with histograms. Sampling and histograms are the two most popular methods for yield estimation in databases. In CAROT, we used a k of 3, classifying yield into low, medium and high, and an n of 100.

Error Metrics: When CAROT is used along with BY cache we use the amount of network savings in the BY cache as a measure to evaluate CAROT. When evaluating the accuracy of CAROT, we consider average relative error.

6.2 Yield Estimation and Cache Performance

Our principal result defines the performance of CAROT in the bypass-yield caching environment. This experiment is conducted against the specific databases for which we will deploy CAROT using workload extracted from those databases. We compare CAROT with the **Optimizer** and the **Sampling** method. We also compare against a **Prescient** estimator, which has *a priori* knowledge of the query result sizes, *i.e.,* a perfect CAROT. The prescient estimator gives us an upper-bound on how well a caching algorithm could possibly perform when the query result sizes are known a-priori.

As a yield estimator for BY caching, CAROT outperforms Optimizer dramatically and approaches the ideal performance of the Prescient estimator. Figure 5 shows the total amount of data sent across the network to serve the entire 1706 GB of the SDSS workload. For any template, the learned model is used for prediction after the 100 queries have occurred. We also include the network savings, *i.e.* the amount of data that was served out of cache, and compare network savings with the ideal performance of Prescient. Based on CAROT's estimates, the BY cache serves the workload saving 532.15 GB when compared with 307.22 GB for the Optimizer. CAROT also compares well with Prescient, reducing network savings by 5% from 558.49 GB to 532.15 GB. (We choose network savings as a metric because it corresponds to cache hit rate.)

Caching results also show the sensitivity of BY caching to accurate yield estimates. Nearly all of the benefit of caching may be lost. The Optimizer loses 45% of network savings and Sampling loses 80%. Sampling is a primitive technique; the Optimizer provides a better point of comparison. However, Sampling does illustrate the sensitivity of BY caching and plays an important role in subsequent experiments.

6.3 Decomposing Workload by Template

To further describe CAROT on the SDSS workload, we take six representative templates that account for 68.7% of the queries (Table 2). The **dimensions in feature vector** indicates the number of attributes that provide information gain for the decision tree. Not all attributes are useful and unused attributes are pruned away by the decision tree algorithm. We also include when indexes are available for attributes referenced by queries in the template. When using the query optimizer, the presence of an index provides better estimates. Notable features of the SDSS workload include: (1) that many important queries do not have indexes and, thus, it may be hard to estimate yield accurately; (2) yield estimation has non-trivial dimensionality, *i.e.,* many attributes are required; and, (3) user defined functions play an important role in the workload. Subsequent experiments examine the performance of CAROT in detail using the queries from these six templates.

6.4 Accuracy of CAROT

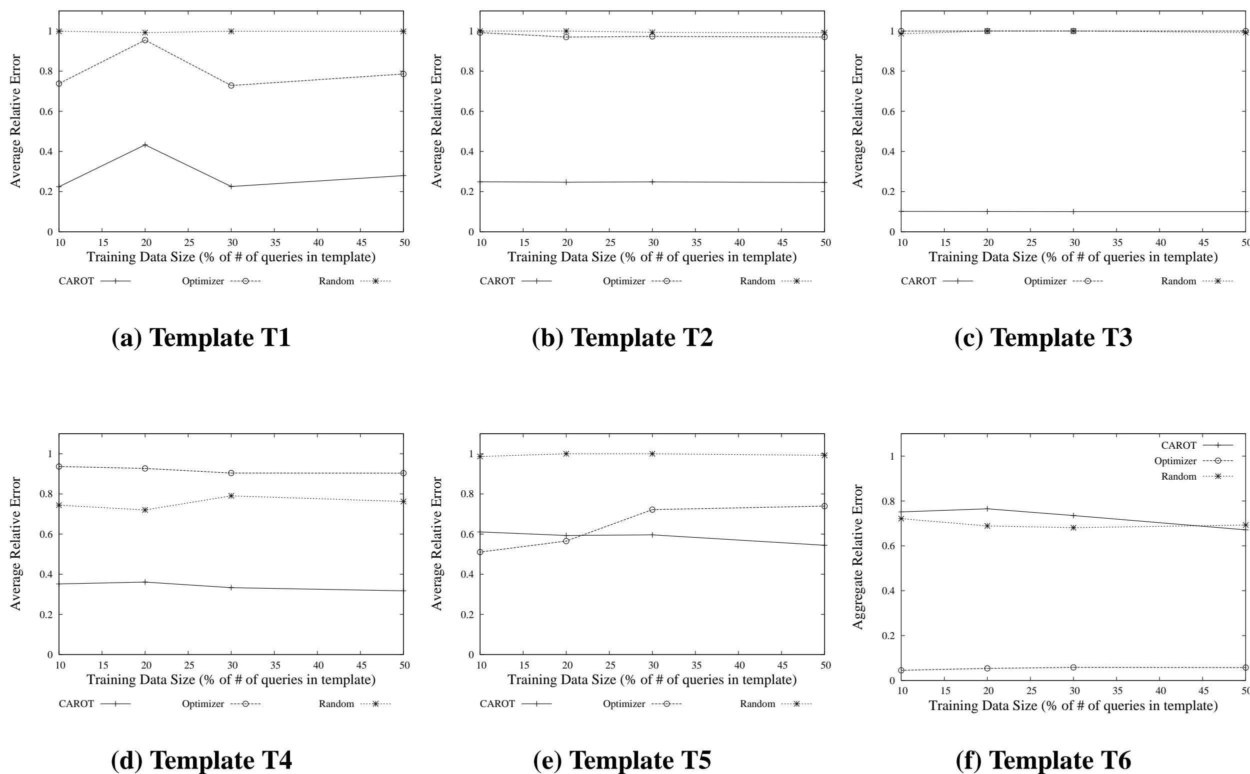
We further describe the performance of CAROT by comparing its accuracy with that of the Optimizer and Sampling within the different templates. In these experiments, the evaluation is static in that we build an *a priori* model using training queries and then use the model to make predictions during testing using testing queries. When comparing accuracy, a static evaluation provides a lower bound on the accuracy of the system. The incrementally updated CAROT system will estimate even more accurately as the model gets updated with test queries. In addition, we keep the distribution of test queries same as those of training queries.

To measure error, we use the absolute error ($\bar{E}_i$) averaged over all queries Q_i that occur in template T_i.

$$\bar{E}_i = \frac{\sum_{q \in Q_i} \mid \rho(q) - \hat{\rho}(q) \mid}{\sum_{q \in Q_i} \rho(q)}, \tag{2}$$

$\rho(q)$ and $\hat{\rho}(q)$ are the actual and estimated yield of a query q. Absolute error corresponds well with our notion of cost and accuracy in caching – network traffic saved, I/O avoided, *etc*.

For range queries, CAROT often estimates yield more accurately than the Optimizer even when an index is available. Figure 6(a) shows results over a two attribute range query on a single table (Template T1). Error arises in the optimizer because of overestimation when selectivity is low and underestimation when selectivity is high. The relative error for CAROT reduces as the size of the training set increases. Sampling performs much worse than CAROT and the query optimizer, because it cannot overcome the large size of the underlying data. The sampled subset represents the true data



(a) Template T1 **(b) Template T2** **(c) Template T3**

(d) Template T4 **(e) Template T5** **(f) Template T6**

Figure 6: Error Results for Templates in the SDSS Workload

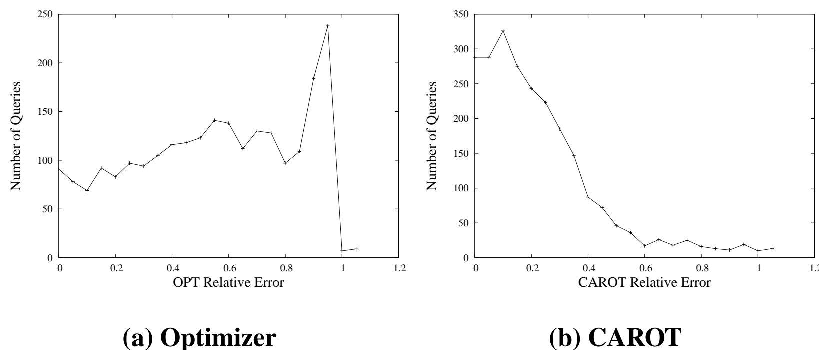
distribution poorly. The hump at 20% occurs for CAROT as it learns unseen data.

CAROT provides the most improvement when operating against user defined functions. Figure 6(b) (Template T2) and 6(c) (Template T3) show function queries against a single table. The opaque nature of function queries make yield estimation difficult for Optimizer and Sampling. The functions in template T2 and T3 look for objects in a circular region and rectangular region, respectively. Often the regions specified are very small. Random sampling suffers because it has not sampled enough objects in the small region specified. The query optimizer performs poorly because it has no semantic information on which to construct an estimate. The Optimizer often returns the same constant for every invocation of a function. Function queries dominate the SDSS workload, as can be seen by the high number of queries in templates T2 and T3.

Scientific queries often use mathematical and bit operators and are quite common in the SDSS workload. Figure 6(d) considers such a query in template T4. Specifically, the predicates in the query add two attributes and compare the result with a constant. The Optimizer predicts poorly as it adds two approximate distributions, thus propagating further errors. Random sampling performs better than the query optimizer, because of the high yield of queries in this template. CAROT learns such distributions precisely as it does not learn the distribution from individual distributions.

CAROT performs well under multi-way joins. Template T5 (Figure 6(e)) is a three-way join, which implies that the accuracy is not limited by the number of tables involved. The same is not true for the optimizer or sampling. Accuracy under multi-way joins is a known limitation of existing selectivity techniques. CAROT starts with a higher error than the optimizer, because of low training data, but rapidly learns and reduces the error significantly.

In the above experiments, CAROT performs well even for small training sets. This is an artifact of user access patterns which possess "locality" properties; they seem to be drawn from a stable distribution. While many templates in the workload possess locality properties, other templates do not, because the distribution of values in their feature space is too disperse. Figure 6(f) shows such a distribution in template T6. The query optimizer gives near accurate estimates for an equality query against an index. Random sampling does not perform well as most of the objects are not sampled, giving a yield of zero for most queries. CAROT does not learn a model from the values in the attribute space, because the objects are accessed randomly from the domain of the queried attributes. Queries in this class are dangerous for CAROT, as it may spend storage and computational effort to no avail. To avoid classification overheads for such queries, we include a set of simple rules to detect equality queries and



(a) Optimizer **(b) CAROT**

Figure 7: CAROT Performs Better than Optimizer Over a Large Fraction of Queries

locate indexes. This relies on schema information, not data access. Such queries are sent to the optimizer (Section 4.3).

To show the performance of CAROT and the query optimizer in more detail (Figure 7), we compare the absolute relative errors of queries chosen from template T1. We chose this template because the presence of index on queried attributes provides a better chance for the optimizer to perform well. By comparing the relative error one can evaluate the performance of the methods on a per query basis. The graphs shows for a given relative error the number of queries witnessed in each method. Optimizer has a large number of queries when the relative error is high unlike CAROT in which most queries have low relative error. This shows that for most queries in the trace, CAROT performs far better than Optimizer.

6.5 Space Utilization and Running Time

The decision tree data structures used by CAROT occupy little space (Table 3). The size of decision trees tends to be in the tens or hundreds of kilobytes. The largest tree overall was that of template T3 at 286 KB, which summarizes the results of over 750,000 queries. This is low requirement in comparison to Optimizer. The total space requirement of max-diff histograms with 5% sampling, where a histogram is constructed for every attribute in a query is 2612 KB, which is 8x more than the space requirement of CAROT. Also, CAROT only grows the decision tree as necessary. The workload for the equality query of template T5 has little information for classification. Thus, CAROT builds a small decision tree, even though there are many queries in the template. However, in the caching system, we would rely on the optimizer estimate for this template, because an index is available.

The height of the decision tree defines the performance of CAROT. Estimating yield takes time $O(h)$ in which h is the height of a decision tree. In practice, decision tree height ranges from one to seven. CAROT also needs to construct the decision tree from training data. Given m attributes and n training examples, decision tree construction takes time $O(mn(logn))$.

Template	Size (KB)	Height
T1	29.03	6
T2	80.43	6
T3	285.66	7
T4	5.62	4
T5	6.12	7
T6	19.15	3

Table 3: Space Performance – Size and Height of Decision Trees

Template	Time (seconds)		
	CAROT	Optimizer	Sampling
T1	3.09	55	211
T2	5.38	345	3290
T3	15.50	890	8000
T4	1.50	0.1	32
T5	0.30	244	522
T6	7.00	0.1	56

Table 4: Time Performance of Estimators

CAROT also exhibits good runtime performance (Table 4). The time to make estimates is negligible – microseconds per query. Because decision trees are small, they easily fit in cache and estimates require only memory accesses. Construction of decision trees and regression functions consumes almost all of time in CAROT, as much as 15.5 seconds for the largest template. The Optimizer incurs substantially more time, except for joins of function templates (T4 and T5) in which it outputs a constant. Sampling performs poorly overall owing to I/O costs, conducting queries on the sampled subset. All experiments were performed on a IBM workstation with 1.3GHz Pentium III processor and 512MB of memory, running Red Hat Linux 8.0.

CAROT and other methods have different execution time profiles. CAROT invests all of its time in model construction and produces estimates nearly instantaneously. The time for the Optimizer and Sampling include only the time to estimate query yields; we did not charge them for building

data structures, sampling, indexing, etc. Thus, time measurements are not directly comparable. While Sampling and the Optimizer have initialization costs, they can be incurred when the data are loaded or updated. In contrast, CAROT prefers to update its data structure occasionally to capture changes in workload. These updates may be performed asynchronously, building new decision trees while the current ones are still in use.

7 Learning Algorithms and Yield Estimation

The data properties dictate the use of appropriate learning algorithms. For yield estimation, an algorithm learns results sizes based on the multi-dimensional parameter space of the template variables. Given the problem of predicting result sizes, regression lends itself naturally to learn the functions on the parameter space. Learning a linear or polynomial multivariate regression function on the parameter space leads to under-fitting, as for complex queries the result sizes are a non-linear function of the parameter space. Therefore we have adopted an initial classification step using decision trees that partitions the parameter space into axis-parallel hyper rectangles. This creates homogeneous regions of the parameter space, in which we use a class-specific regression functions to estimate the yield. An alternative approach is to use regression trees [Mitchell 1997], which are like decision trees but can directly output result sizes obviating the need to learn a subsequent regression function. However, when compared with the combination of classification (with decision trees) and regression, on our datasets, the constructed regression trees were larger and took longer to update. Also while they were slightly more accurate, they took much longer to learn.

We opted not to use other learning techniques for a variety of reasons. Bayes techniques [Mitchell 1997] assume conditional independence among input variables. This assumption is not true for astronomy data and scientific data sets in general. In fact, this assumption is also made by the optimizer and leads to its inaccuracy. The high-dimensionality of the input data renders some techniques, such as support vector machines [Mitchell 1997], inefficient. Finally, nearest neighbor techniques [Mitchell 1997] are also very slow for prediction time, as computing nearest neighbors for every query point is time consuming.

8 Conclusions and Future Work

In CAROT, we have presented a novel approach to yield-estimation using the data mining techniques of classification and regression. Estimation is query-based and adaptive: it builds a distribution on classes of queries and their results and uses further queries to refine and update the distribution. The estimation model suits the stringent access and performance requirements of bypass-yield caching. We have evaluated our approach on workloads from the Sloan Digital Sky Survey. Experimental results verify the efficacy and accuracy of CAROT.

In the future, we are interested in the applicability of CAROT to non-scientific workloads. Many of CAROT's benefits derive from properties of the SDSS workload. Metadata are compact because there are very few different classes of queries. The high-dimensionality of data and complex joins of the workload are easily learned by CAROT and make optimizer estimates less accurate. Also, astronomy databases are static: no fine-grained updates, only major data releases. Issues that arise in other workloads include the management of a large number of templates and the consistency of CAROT's metadata when updating databases. Inclusion of such features will broaden the applicability of CAROT to various other workloads. It is to be noted, however, that the CAROT is a general purpose estimation tool. Workload properties such as existence of few templates enhance the scalability of CAROT but do not undermine its ability to estimate accurately.

Merging related templates offers a promising approach to reducing the number of templates induced by a workload. Templates grow with the number of different queries on the same relation. For templates that share some attributes and describe similar distributions, a merged, more general template reduces metadata without compromising accuracy. However, creating general templates by merging results in *missing values* within feature vectors. Thus, we are applying studies of missing values in decision trees to CAROT [Zhang et al. 2005]. For example, we want to understand how the yield of a join query can be interpreted from the two existing templates of the corresponding relations. We are also working on more complex queries, such as nested queries and queries that fetch images.

We are also interested in integrating CAROT into database query optimizers. CAROT provides accurate estimates for queries that have challenged optimizers in the past, specifically, user-defined functions and multi-dimensional queries. The key feature of CAROT for caching is its ability to provide estimates without access to data, *i.e.,* remotely from databases. However, this does not preclude its use in databases. It seems that an optimizer could deploy CAROT to handle the special cases in which it cannot provide good estimates. This use would mirror CAROT's use of the optimizer for queries in which CAROT has not yet learned a yield distribution or when indexes are available.

Acknowledgments

The authors sincerely thank Jim Gray for his help with the selectivity estimation methods proposed for query optimizers. The authors thank Amitabh Chaudhary for giving us the idea of the multidimensional example. We thank Xiodan Wang for his help with the SDSS workload. Finally, we would like to thank our anonymous reviewers who helped us improve the presentation of this paper.

References

ABOULNAGA, A., AND CHAUDHURI, S. 1999. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 181–192.

AMBITE, J. L., AND KNOBLOCK, C. A. 1998. Flexible and scalable query planning in distributed and heterogeneous environments. In *Conference on Artificial Intelligence Planning Systems*, 3–10.

AMIRI, K., PARK, S., TEWARI, R., AND PADMANABHAN, S. 2003. Scalable template-based query containment checking for Web semantic caches. In *Proceedings of the International Conference on Data Engineering*, 493–504.

BRUNO, N., AND CHAUDHURI, S. 2002. Exploiting statistics on query expressions for optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 263–274.

BRUNO, N., CHAUDHURI, S., AND GRAVANO, L. 2001. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

CAO, P., AND IRANI, S. 1997. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technology and Systems*.

CHAKRABARTI, K., GAROFALAKIS, M. N., RASTOGI, R., AND SHIM, K. 2000. Approximate query processing using wavelets. In *Proceedings of the International Conference on Very Large Data Bases*, 111–122.

CHEN, C. M., AND ROUSSOPOULOS, N. 1994. Adaptive selectivity estimation using query feedback. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 161–172.

CHRISTODOULAKIS, S. 1983. Estimating block transfers and join sizes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 40–54.

DREWRY, M., CONOVER, H., MCCOY, S., AND GRAVES, S. 1997. Meta-data: quality vs. quantity. In *Second IEEE Metadata Conference*.

FEDCACHE. FedCache: Proxy Caching in Wide-Area Scientific Federations, http://hssl.cs.jhu.edu/fedcache.

FUJIWARA, H., AND IWAMA, K. 2002. Average-case competitive analyses for ski-rental problems. In *Intl. Symposium on Algorithms and Computation*.

GUNOPULOS, D., KOLLIOS, G., TSOTRAS, V., AND DOMENICONI, C. 2000. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the 19th ACM-SIGMOD Intl. Conference on Management of Data*.

HAMERLY, G., AND ELKAN, C. 2003. Learning the k in k-means. In *NIPS*.

HARANGSRI, B., SHEPHERD, J., AND NGU, A. H. H. 1997. Query size estimation using machine learning. In *Database Systems for Advanced Applications*, 97–106.

IOANNIDIS, Y. E. 2003. The history of histograms (abridged). In *Proceedings of the International Conference on Very Large Data Bases*, 19–30.

LIM, L., WANG, M., AND VITTER, J. S. 2005. CXHist : An on-line classification-based histogram for XML string selectivity estimation. In *Proceedings of the International Conference on Very Large Data Bases*, 1187–1198.

LIPTON, R., AND NAUGHTON, J. 1995. Query size estimation by adaptive sampling. *Journal of Computer and System Science 51*, 18–25.

LUO, Q., AND NAUGHTON, J. F. 2001. Form-based proxy caching for database-backed web sites. In *Proceedings of the International Conference on Very Large Data Bases*, 191–200.

LUO, Q., AND XUE, W. 2004. Template-based proxy caching for table-valued functions. In *Proceedings of the International Conference on Database Systems for Advanced Applications*, 339–351.

MALIK, T., SZALAY, A. S., BUDAVRI, A. S., AND THAKAR, A. R. 2003. SkyQuery: A WebService Approach to Federate Databases. In *Proceedings of the Conference on Innovative Data Systems Research*.

MALIK, T., BURNS, R. C., AND CHAUDHARY, A. 2005. Bypass caching: Making scientific databases good network citizens. In *Proceedings of the International Conference of Data Engineering*, 94–105.

MITCHELL, T. 1997. *Machine Learning*. McGraw Hill.

OLSTON, C., AND WIDOM, J. 2002. Best-effort cache synchronization with source cooperation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 73–84.

OLSTON, C., LOO, B., AND WIDOM, J. 2001. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 355–366.

OSQ. The Open Skyquery, http://www.openskyquery.net.

PELLEG, D., AND MOORE, A. 2000. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, Morgan Kaufmann, San Francisco, 727–734.

POOSALA, V., AND IOANNIDIS, Y. E. 1996. Estimation of query-result distribution and its application in parallel-join load balancing. In *The VLDB Journal*, 448–459.

POOSALA, V., IOANNIDIS, Y. E., HAAS, P. J., AND SHEKITA, E. J. 1996. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 294–305.

QUINLAN, J. R. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

SDSS. The Sloan Digital Sky Survey, http://www.sdss.org.

SELINGER, P., ASTRAHANAND, M., CHAMBERLIN, D., A.R.LORIE, AND PRICE, T. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

SERAFINI, L., STOCKINGER, H., STOCKINGER, K., AND ZINI, F. 2001. Agent-based query optimisation in a grid environment. In *Proceedings of IASTED International Conference on Applied Informatics*.

SHETH, A. P., AND LARSON, J. A. 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys 22*, 3, 183–236.

STILLGER, M., LOHMAN, G. M., MARKL, V., AND KANDIL, M. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the International Conference on Very Large Data Bases*, 19–28.

SZALAY, A. S., GRAY, J., THAKAR, A. R., KUNSZT, P. Z., MALIK, T., RADDICK, J., STOUGHTON, C., AND VANDENBERG, J. 2002. The SDSS SkyServer: Public access to the Sloan Digital Sky Server data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 570–581.

VITTER, J. S., AND WANG, M. 1999. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 193–204.

W. SUN, Y. LING, N. R., AND DENG, Y. 1993. An instant and accurate size estimation method for joins and selection in a retrieval-intensive environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 79–88.

WEKA. The Weka Machine Learning Project. http://www.cs.waikato.ac.nz/ ml/index.html.

ZHANG, S., QIN, Z., LING, C. X., AND SHENG, S. 2005. Missing is useful: Missing values in cost-sensitive decision trees. In *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, 1689 – 1693.