Bypass Caching: Making Scientific Databases Good Network Citizens *

Tanu Malik, Randal Burns
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
{tmalik, randal} @cs.jhu.edu

Amitabh Chaudhary
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
Amitabh.Chaudhary.1@nd.edu

Abstract

Scientific database federations are geographically distributed and network bound. Thus, they could benefit from proxy caching. However, existing caching techniques are not suitable for their workloads, which compare and join large data sets. Existing techniques reduce parallelism by conducting distributed queries in a single cache and lose the data reduction benefits of performing selections at each database. We develop the bypass-yield *formulation of caching, which reduces network traffic in wide-area database federations, while preserving parallelism and data reduction. Bypass-yield caching is* altruistic*; caches minimize the overall network traffic generated by the federation, rather than focusing on local performance. We present an adaptive, workload-driven algorithm for managing a bypass-yield cache. We also develop on-line algorithms that make no assumptions about workload: a k-competitive deterministic algorithm and a randomized algorithm with minimal space complexity. We verify the efficacy of bypass-yield caching by running workload traces collected from the Sloan Digital Sky Survey through a prototype implementation.*

1. Introduction

An increasing number of science organizations are publishing their databases on the Internet, making data available to the larger community. Applications such as Sky-Query [37], PlasmoDB [31], and Distributed Oceanographic Data System (DODS) [14] use the published archives for comprehensive experiments that involve merging, joining, and comparing Gigabyte and Terabyte datasets. As these data-intensive scientific applications increase in scale and number, network bandwidth constrains the performance of all applications that share a network.

We are particularly interested in the scalability and network performance of SkyQuery [27]. SkyQuery is the

mediation middleware used in the World Wide Telescope (WWT) – a virtual telescope for multi-spectral and temporal experiments. The WWT is an exemplar scientific database federation, supporting queries across vast amounts of freely-available, widely-distributed data [15]. The WWT faces an impending scalability crisis. With fewer than 10 sites, network performance limits responsiveness and throughput already. We expect the federation to expand to more than 120 sites in 2006.

While caching is the principal solution to scalability and performance, existing database caching solutions fail to meet the needs of scientific databases. Caching is a dangerous technology because it can reduce the parallelism and data filtering benefits of database federations. Thus, caching must be applied judiciously. A query against a federation is divided into sub-queries against member sites, which are evaluated in parallel. Parallel evaluation brings great computational resources to bear on experiments that are initiated from the weakest of computers. Caching can reduce parallelism by moving workload from many databases to few caches. Running queries at the databases also filters results [4], producing compact results from large tables. Many scientific queries operate against a large amount of data. Bringing the large data into cache and computing a small result can waste an arbitrarily large amount of network bandwidth.

The primary goal in current database caching solutions [3, 18, 22] is to maximize hit rate and minimize response time for a single application. Minimizing network traffic is a secondary goal. Organizations have no direct motivation to reduce network traffic because they are not charged by the amount of bandwidth they consume. However, it is imperative for data-intensive applications to focus on being good "network citizens" and using shared resources conscientiously. If not, the workloads generated by these applications will make them unwelcome on public networks.

We propose bypass-yield caching, an altruistic caching framework for scientific database workloads. As its principal goal, it adopts network citizenship: caching data in order to minimize network traffic. Bypass-yield caching profiles workload to differentiate between data objects for which caching saves network bandwidth and those which should not be cached. The latter are routed directly to the back-end

database servers. Our experiments show that this framework leads to an overall network traffic reduction.

1.1. Our Contributions

In this paper, we model bypass-yield caching as a rent-to-buy problem and develop economic algorithms that satisfy the primary goal of network traffic reduction. The algorithms also meet the requirements of proxy database caching, specifically, database independence and scalable metadata. We implement these algorithms in the World-Wide Telescope.

We introduce the concept of *yield* employed in bypass-yield caching. The yield model differs from classical caching systems, such as page model and object model caching, in that it considers the amount of data delivered to an application on a per request basis. In the page model, a cache contains objects of a fixed size (pages) and a "cache hit" occurs when an application reads an entire object. Memory hierarchies and operating systems use the page model [21, 28, 30, 44]. The object model expands upon the page model to account for variable object sizes and non-uniform distances to data sources. Again, a cache hit involves accessing the entire object in cache. The object model applies to Web caching systems [6, 18, 22–24, 29, 42, 43] and distributed file stores [17]. The yield model follows the object model in that objects vary in size and each has its own fetch cost. However, applications that access objects in the cache see variable benefits, depending upon how many bytes of data the request returns. In a yield cache, queries may return partial results based on selectivity criteria, or they may return an aggregate computed over an object.

We define yield-sensitive metrics for caching. Specifically, we develop the *byte-yield hit rate* (*BYHR*), which generalizes the concept of hit rate in the yield model. *BYHR* measures the rate at which a cached object reduces network traffic normalized to its size (amount of space consumed in a cache). *BYHR* can be used to evaluate the utility of a cache and cache management algorithms. We employ the metric for eviction and loading decisions in our algorithms.

The yield model leads naturally to our formulation of bypass caching in which cache management algorithms make economic decisions that minimize network traffic. We choose between loading an object and servicing queries for that object in the cache versus *bypassing* the cache and sending queries to be evaluated at sites in the federation. Network bandwidth is the currency of this economy. An algorithm invests network traffic to load an object in the cache in order to realize long term savings. For any given request, loading an object uses more network bandwidth than does evaluating the query and shipping query results. The bypass concept is closely related to hybrid shipping [39] and similar concepts are employed by query optimizers [41].

Based on bypass-yield metrics, we develop a workload-driven, rate-based algorithm for cache management. The algorithm profiles queries against data objects, evaluating the rate of network traffic reduction. We make the bypass versus load/eviction decision when a query occurs for an object not present in the cache. The algorithm compares expected rate of savings of an outside object with the minimum current rate of savings of all objects in the cache. Objects outside the cache are penalized by the network cost to fetch them from the database federation. Queries to objects for which the rate does not exceed the minimum are bypassed. Aging and pruning techniques allow the algorithm to adapt to workload shifts and to keep metadata compact and computationally efficient.

The rate-based algorithm works well in practice, but lacks some desirable theoretical properties. In particular, it depends upon some degree of workload stability; it uses prior workload as an indicator of future accesses. Also, it maintains metadata, in the form of query profiles, for objects in the federation.

We address the theoretical shortcomings of the rate-based algorithms through on-line algorithms that make no assumptions about workload patterns. We present a k-competitive algorithm that uses the rent-to-buy principle to load objects into the cache only after bypass queries have generated network traffic equal to the load cost. We also present a randomized version of the algorithm with minimum space complexity. It chooses to load objects into the cache with a probability proportional to the yield of the current query.

We have implemented all of the above algorithms and experimentally evaluate their performance using a Terabyte astronomy workload collected from the Sloan Digital Sky Survey [36]. We also compare their performance against competitive in-line caching (without bypass), optimal static caching, and execution without caching. Results indicate that bypass-yield algorithms approach the performance of optimal static caching.

Experimental results also address the question of what to cache for scientific workloads. We compare caching tables, columns, and semantic (query) caching. Semantic caching is attractive for database federations because it preserves their filtering benefits. In fact, semantic caching lies outside the bypass-yield framework; *i.e.*, bypass-yield depends on query evaluation within the cache. However, we find that astronomy workloads do not exhibit the query reuse and query containment upon which semantic caching relies [25]. Rather, astronomy workloads exhibit schema reuse, conducting queries with similar schema against different data. For example, a common query iterates over regions of the sky looking for objects with specific properties. Thus, we employ tables and columns as objects to cache in the bypass-yield framework.

2. Related Work

All caching systems address vital issues, such as cache replacement, cached object granularity, cache consistency,

and cache initialization. In this section, we review related work on cache replacement algorithms, the choice of object to cache, and the concept of bypass caching.

In the general paging problem, pages have varying sizes and fetch costs. The goal is to maintain a cache of fixed size so as to minimize the total fetch cost over cache misses [3]. Paging, as used in memory and buffer management systems [10, 11, 19, 44], caches pages that have uniform size and cost. Cache replacement policies that are commonly used in such environments are LRU, LFU, and LRU-K [30], among others. These algorithms use the *single basic* property of the reference stream [22] in order to minimize the number of page faults.

The Greedy-Dual (GD) algorithm [44] introduces variable fetch costs for pages of uniform size. The Greedy-Dual-Size (GDS) algorithm [6, 18] extends GD to the object model in which objects have variable size and fetch cost. On each access to an object, GDS assigns it a *utility value* equal to its *cost/size* ratio. The utility value ages over time, keeping objects with high temporal locality in cache. GDS has been found to work well in practice [9, 18]. The public-domain Squid Web proxy [42] incorporates a modified version of GDS.

Proxy database caches also employ the object model with variable size and fetch cost. Objects may be query results [33–35] or database objects, such as relations, attributes, and materialized views [7, 32]. Much attention has been paid to cache organization and integration with query optimization in different application environments [16, 20]. However, these systems use simple policies for cache eviction, such as LRU, LFU, LFF, or heuristic combinations of these simple policies.

Many caching algorithms use the reference information in the workload to make better decisions. LRU-K [30] extends LRU to incorporate information for the last K references to popular pages. For database disk buffering, LRU-K outperforms conventional buffering algorithms in discriminating between frequently and infrequently referenced pages. GDSP [22] extends GDS to include a frequency count in the utility measure. Their results show that it outperforms GDS empirically on HTTP traces. Our rate-based algorithm uses frequency count similar to GDSP for all objects in the reference stream, not just those in the cache currently.

Irani [18] gives algorithms for what can be considered a bypass cache in the object model. They mention that the option of being able to bypass a request, *i.e.*, serve a miss without first bringing the requested object into the cache, does not help in reducing the cost of a request sequence. This, however, is not true for database caches in which the size of the query result may be much smaller than the size of the object. Scheuermann *et al* [35] take advantage of variable query result sizes in a cache admission policy that improves performance by 32%. Their objective is to minimize the execution cost of a query sequence. Neither apply di-
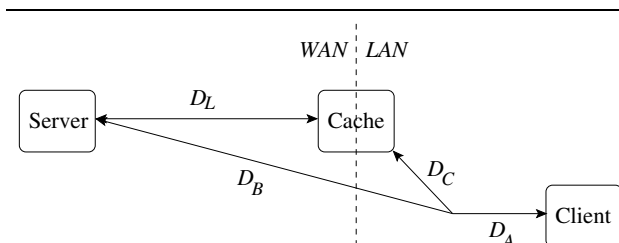

Figure 1. Network flows in a client/server pair using bypass caching.

rectly to our objective of minimizing network cost.

There have been several industry initiatives to develop caching systems for relational objects [1, 40] and for dynamic materialized views that may overlap with each other [2]. The cache policies used in these systems are simple compared to hierarchical and widely-distributed systems, such as Squid [42], but they do underline the benefits of caching in database systems.

In this paper, we extend the Web caching problem and its solutions to incorporate the concept of yield. We are concerned not only with objects that have variable fetch costs and sizes, but also with queries on these objects that return results of variable size. Through bypass-yield caching and the presented algorithms, we extend the benefits of caching in the Web environment to the (more complex) database environment.

3. The Bypass-Yield Caching Model

Our formulation of bypass caching includes both query scheduling and network citizenship. In a database federation, we collocate a caching service with mediation middleware. When the mediator receives a query against the federation, it divides it into sub-queries. Each sub-query is a query against a single site in the federation. The cache evaluates whether to service each sub-query locally, loading data into the cache, versus shipping each sub-query to be evaluated at a server in the federation. We term the latter a *bypass query*, because the query and its results circumvent the cache. A bypass cache is not a semantic cache [12] and does not attempt to reuse the results of individual queries.

Caches elect to bypass queries in order to minimize the total network cost of servicing all the queries. We assume that the cache resides near the client on the network and try to minimize data flow from the servers to the cache(s) and client(s) combined. Because each cache acts independently, the global problem can be reduced to individual caches (Figure 1). At this time, we do not consider hierarchies of caches or coordinated caching within hierarchies. The network traffic to be minimized (WAN) is the bypass flow from a server directly to a client D_B, plus the traf-

fic 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.*, $D_A = D_B + D_C$, where D_C is the traffic from the queries served out of the local cache. The local area network is not a shared resource and is scalable. LAN traffic does not factor into network citizenship.

We develop the concept of *yield* to measure the effectiveness of a bypass cache in the database environment. The yield of a query is the number of bytes in the query result. It measures both the network cost of a bypass query and the network savings of a query served in the cache.

When composed with object sizes and load costs, yield leads to metrics for bypass-yield caching. In the byte-yield hit rate (*BYHR*), we define one such metric. *BYHR* measures the benefit of caching an object, *i.e.*, the rate of network bandwidth reduction per byte of cache. The metric is evaluated based on workload and object properties. Thus, every object in the system has a *BYHR*, regardless of whether it is being cached or not. *BYHR* is defined as

$$BYHR_i = \sum_j \frac{p_{i,j} y_{i,j} f_i}{s_i{}^2}, \qquad (1)$$

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. *BYHR* can be decomposed into two components. The first arises from yields of the different queries, $\sum_j p_{i,j} y_{i,j}/s_i$, and measures the potential benefit of caching an object. That is, the number of bytes delivered to the application $\sum_j p_{i,j} y_{i,j}$ normalized to the amount of cache space s_i. This component prefers objects for which the workload would yield more bytes out of the cache, and, thus, more network savings per unit of cache space occupied. The second component, f_i/s_i, describes the variable fetch cost f_i of different objects from different sources, again, normalized by the object size s_i. This component helps evict objects with lower fetch costs, because re-loading these objects into the cache will be less expensive.

Often, the fetch cost will be proportional to the object size, $f_i = cs_i$ for some constant c. This is true when (1) caching objects from a single server, (2) caching objects from multiple collocated servers, or (3) when networks have uniform performance. The proportional assumption also relies on networks exhibiting linear cost scaling with object size, which is true for TCP networks when the transfer size is substantially larger than the frame size [38]. For these environments, we use a simplification of *BYHR*, called the byte-yield utility, defined as

$$BYU_i = \sum_j \frac{p_{i,j} y_{i,j}}{s_i}. \qquad (2)$$

BYHR generalizes previous cache utility metrics in simpler caching models. Page model caching systems use hit rate to evaluate the cache management policies. *BYU* degenerates to hit rate for objects of a constant size, with yield equal to the object size. In the object model, *BYHR* degenerates to GDSP for yield equal to the object size. The generalization extends to algorithms also; *BYHR* results in algorithms that achieve the same bounds as competitive paging algorithms in the page [44] and object [18] models.

A key challenge in employing the proposed metrics lies in evaluating the probabilities of queries. One approach estimates probabilities based on observed workload access patterns. We employ such estimates in the rate-based algorithm and develop techniques for aging and pruning. Aging allows estimates to adapt to changing workloads and pruning limits the amount of metadata. A different approach creates algorithms that perform well on any access pattern. Rather than predicting or estimating probabilities, algorithms use economic principles to manage cache contents. We take this approach in our on-line algorithms.

4. The Rate-Profile Algorithm

In this section, we develop the Rate-Profile algorithm using the byte-yield utility (*BYU*) metric. The algorithm can be trivially extended to use byte-yield hit rate (*BYHR*) for multiple sources on non-uniform networks. We describe (1) cache eviction policies based on the performance of objects in the cache and (2) the bypass versus cache load decision based on comparison of objects not in the cache with the current cache state. When combined, these policies form an algorithm for managing a bypass-yield cache.

The Rate-Profile algorithm compares the expected network savings of items not in the cache against the current performance of cached items. Both quantities are expressed as rate of savings: bytes (on the network) per unit time per unit byte (of cache space occupied). Time is relative and measured in number of queries in a workload, not seconds.

Rate-Profile estimates access probabilities in *BYU* using workload as a predictor. Recency and frequency are important aspects of the estimates. For objects in the cache, we maintain frequency counts, which are used as a probability estimate. We enforce recency by only evaluating frequency over the cache lifetime. For objects not in the cache, the issue of recency is more complex. We use heuristics to divide the past into *episodes*, which represent clustered accesses to a data object. Within each episode, we use frequency counting to estimate access probability; the division into episodes enforces recency. When computing a rate of savings estimate for items not in the cache, we consider all episodes, weighing the contributions of recent episodes more heavily.

4.1. Eviction

The utility of an object in the cache is the measured rate of network savings realized from queries against that object over its cache lifetime. We construct a continuous time metric that measures utility, which we call a rate profile (*RP*).

The *RP* of object o_i is defined as

$$RP_i(t) = \frac{\sum_j y_{i,j}}{(t - t_i)s_i} \qquad (3)$$

in which an object o_i of size s_i is accessed by queries Q_i, with each query $q_{i,j} \in Q_i$ yielding $y_{i,j}$ bytes. Each of these queries occur between time t and t_i, where t can be evaluated at all times during which an object is cached and t_i is the time at which the object was loaded into the cache. The rate profile measures the *BYU* of an object over its cache lifetime. Accesses to an object increase the *RP* and time decays it. Rate profiles are compact and easy to update in that they simplify complex access patterns into an average rate of network savings. For each query serviced in the cache, we track and sum the yield, allowing us to evaluate the rate profile at any time.

Rate profiles of different items are compared to select "victims": items to be evicted from the cache. The item with the lowest rate profile has the lowest rate of savings. Thus, to create space for new objects in the cache, we discard items with the lowest current *RP*. Because time decays the *RP*, unused items age out of the cache. The *RP* does not retain the specific times of accesses and, thus, is not weighted towards recency within the scope of an object's cache lifetime. However, a lower rate does represent lower average savings.

4.2. The Bypass Decision

For objects not in cache, we compute utility based on past performance, which estimates future network savings. In the load-adjusted rate (*LAR*), we construct a metric based on *BYU* that estimates the utility of an object were it to be loaded into the cache. The *LAR* is expressed in savings rate: the same units as cache *RP*s. Thus, the *LAR* may be used to compare directly the expected performance of an object not in the cache against the current performance of the cache contents. In the *LAR*, we account for episodes of queries and composing and aging the contributions of all episodes.

To compute the *LAR*, we require some intermediate quantities. Items incur a network cost to be loaded into the cache, which reduce the total network savings. We account for this by reducing the rate profile by the load cost of an object, constructing a load-adjusted rate profile for an individual episode

$$LARP_{i,e}(t) = \frac{\sum_j y_{i,j}}{(t - t_S(i,e))s_i} - \frac{f_i}{s_i} \qquad (4)$$

in which and $t_S(i,e)$ is the start time of the current episode and t is any time in episode e subsequent to $t_S(i,e)$. This quantity is a profile, a continuous time metric, and must be distilled into a single value. For each episode, we take the maximum value, which represents the maximum rate of savings that would have been realized were the object to be

cached for that episode. The load-adjusted rate (of savings) for an episode is

$$LAR_{i,e} = \max_{t \in \{t_S(i,e)+1, t_E(i,e)\}} LARP_{i,e}(t) \qquad (5)$$

in which $t_E(i,e)$ is the end time of episode e. The maximum value describes the balance point between network savings overcoming the initial load cost and reduced usage of the object decreasing its utility. Finally, we consider contributions over all episodes to generate an expected rate of savings

$$LAR_i = \frac{\sum_{e \in E_i} LAR_{i,e} w_e}{\sum_{e \in E_i} w_e} \qquad (6)$$

in which w_e is a function that weights the contributions of episode e drawn from all episodes E_i of object o_i.

To make the bypass decision when the cache receives a query, we compare the *LAR* of the requested object against the minimum *RP*s of items in the cache. If enough cached objects have lower *RP*s (to make space for the requested object), the requested object will be loaded into the cache. Otherwise, the query will be bypassed.

Rate-Profile employs simple economic principles. It invests in the future savings when loading an object and incurs a load penalty when the expected savings of an object not in the cache exceeds the current savings of objects in the cache. However, when using the *RP* to evaluate objects already in the cache, we do not include a load penalty, because it is a sunk cost. This ensures that the cache is conservative in its evictions, which is an important aspect of algorithms in the bypass-yield model. Objects must reside in the cache long enough to recover the load investment. All of this decision making is based on comparing expected versus current network savings using a single currency.

4.3. Episodes

We employ simple heuristics that divide the workload against an object into disjoint episodes. Each episode represents a clustered set of accesses to an object. There are hazards in choosing episodes incorrectly. If episodes are too long, then the utility of an object gets reduced by averaging over too long an interval. If episodes are too short, then the object does not get used enough to overcome the load penalty. An episode begins on the first access to an object and we terminate the current episode and start a new episode when either

1. $LARP_{i,e} < c \cdot LAR_{i,e}$ or
2. the object has not been accessed during the last k queries.

In our experiments, we chose $c = 0.5$ and $k = 1000$. The first rule extends episodes as long the rate increases and allows for some decrease in rate in order to survive short idle periods between bursts of traffic. The second rule ensures that the episodes of lightly used objects do not last

for long periods, observing that the rate will always be increasing until the load penalty has been overcome, *i.e.*, until $LARP_{i,e} > 0$. The parameters of these heuristics have not been tuned carefully, nor do we support that this is the only or best technique for dividing workload. Our experience dictates that episodes are mandatory to deal with bursts in workload and that our results are robust to many parameterizations.

5. On-line Algorithms

We present OnlineBY and SpaceEffBY, the second and third in our suite of algorithms for bypass-yield caching. OnlineBY achieves a minimum level of performance for any workload. In particular, we show its cost is always at most $O(\lg^2 k)$ times that of the optimal algorithm, where k is the ratio of the size of the cache to the size of the smallest possible object in the cache. Further, to achieve this performance, OnlineBY does not need to be trained with a representative workload. However, we expect OnlineBY to under-perform Rate-Profile in practice, because it forgoes workload information.

SpaceEffBY is a space efficient algorithm. Both Rate-Profile and OnlineBY need to store information for objects in the federation, whether they are in the cache or not. This may prove to be impractical. SpaceEffBY uses the power of randomization to do away with the need to store object metadata. It has, however, no accompanying performance guarantees.

OnlineBY is an amalgamation of algorithms for two online sub-problems: (1) the on-line ski rental problem and (2) the bypass-object caching problem. The next subsection describes these sub-problems and their known algorithms. The subsection after that describes OnlineBY and proves the bound on its performance. Finally, there is a subsection describing SpaceEffBY.

5.1. Related Sub-problems

On-line ski rental is a classical rent-to-buy problem [13]. A skier, who doesn't own skis, needs to decide before every skiing trip that she makes whether she should rent skis for the trip or buy them. If she decides to buy skis, she will not have to rent for this or any future trips. Unfortunately, she doesn't know how many ski trips she will make in future, if any. This lack of knowledge about the future is a defining characteristic of on-line problems [5]. A well known on-line algorithm for this problem is rent skis as long as the total paid in rental costs does not match or exceed the purchase cost, then buy for the next trip. Irrespective of the number of future trips, the cost incurred by this on-line algorithm is at most twice of the cost incurred by the optimal offline algorithm.

If there was only one object to cache, the bypass-yield problem would be nearly identical to on-line ski rental. By-passing a query corresponds to renting skis and loading the object into the cache corresponds to buying skis. The

one difference is that renting skis always costs the same, whereas the yield from queries differs. However, the same algorithm applies to the bypass-yield problem, again at cost no more than twice optimal.

Bypass-object caching can be viewed as a restricted version of the bypass-yield caching in which queries are limited to those that return a single object in its entirety. Formally, in bypass-object caching, we receive a request sequence $\sigma_{\text{obj}} = o_1, \ldots, o_n$ for objects of varying sizes. Let the size of object o_i be s_i. If o_i is in the cache, we *service* the request at cost zero. Otherwise, we can either (1) *bypass* the request to the server or (2) first *fetch* the object into the cache and then service the request. Both cases incur cost f_i. In the former, the composition of the cache does not change. In the latter, if the cache does not have enough space to store o_i, we evict some objects from those currently cached to create the required space. The objective is to respond to each request in a manner that minimizes the total cost of servicing the sequence σ_{obj} without knowledge of future requests.

Irani [18] gives an $O(\lg^2 k)$-competitive algorithm for bypass-object caching. She calls it optional multi-size paging under the byte model. Recall that an on-line algorithm $\mathcal{A}_{\text{obj}}$ is said to be α-competitive if there exists a constant b such that for every finite input sequence σ

$$\text{cost}(\mathcal{A}_{\text{obj}} \text{ on } \sigma) \leq \alpha \cdot \text{cost}(\text{OPT on } \sigma) + b$$

in which OPT is the offline optimal. Again, k is the ratio of the size of the cache to the size of the smallest possible object.

OnlineBY extends the algorithm for bypass-object caching to the bypass-yield caching problem. It does so by running an instance of the on-line ski rental algorithm for every object that is being queried. When enough queries for an object have arrived such that the cumulative yield matches or exceeds the size of the object, OnlineBY treats the situation as if a request for the entire object arrives in bypass-object caching. The next subsection describes OnlineBY formally and shows that it is $O(\lg^2 k)$-competitive.

5.2. OnlineBY

OnlineBY is an on-line algorithm for the bypass-yield caching problem. OnlineBY receives a query sequence $\sigma = q_1, \ldots, q_n$. Each query q_j refers to a single object o_i and yields a query result of size $y_{i,j}$. If o_i is in the cache, we *service* the query at cost zero. Otherwise, we can either (1) bypass the query to the server at cost $c(q_j)$ or (2) *fetch* object o_i into the cache at cost f_i and service the query in cache. We define $c(q_j)$ equal to $(y_{i,j}/s_i) \cdot f_i$, in which s_i is the size of o_i. In the former case, the composition of the cache does not change. In the latter case, the cache evicts objects, as necessary, to create storage space for o_i. The objective is to respond to each query in a manner that min-

```
OnlineBY (q_j)
    /*  q_j is the next query in the input sequence.
        q_j refers to object o_i and has yield y_{i,j}.
        For all i, BYU_i is initially set to 0.
        A_obj is an algorithm for bypass-object caching. */

    BYU_i ← BYU_i + y_{i,j}/s_i.
    If (BYU_i ≥ 1)
        BYU_i ← BYU_i − 1.
        o_i is generated as the next input for A_obj.
    The cache is maintained according to A_obj.
    If (o_i is in the cache)
        Service q_j from the cache.
    Else
        Bypass q_j to the server.
```

Figure 2. The OnlineBY Algorithm.

imizes the total cost of servicing the sequence σ without knowledge of future queries.

OnlineBY uses any on-line algorithm for the bypass-object caching $\mathcal{A}_{\mathrm{obj}}$ as a sub-routine. This generates a family of on-line algorithms for bypass-yield caching based on different on-line algorithms for bypass-object caching. OnlineBY is described in Figure 2. The algorithm employs the byte-yield utility metric (BYU_i, Section 4). As with Rate-Profile, on-line algorithms can be extended trivially to *BYHR*. OnlineBY *maintains* a cache according to what $\mathcal{A}_{\mathrm{obj}}$ does. In other words, OnlineBY loads and evicts the same objects as $\mathcal{A}_{\mathrm{obj}}$ at the same times (in response to the object sequence presented by OnlineBY).

The main result of this section is the following.

Theorem 5.1 *For every α-competitive on-line algorithm $\mathcal{A}_{\mathrm{obj}}$ for bypass-object caching,* OnlineBY *creates a corresponding on-line algorithm for bypass-yield caching that is $(4\alpha + 2)$-competitive.*

Corollary 5.2 *There exists an on-line algorithm for bypass-yield caching that is $O(\lg^2 k)$-competitive, where k is the ratio of the size of the cache to the size of the smallest object.*

Proof: The corollary follows from the on-line algorithm given by Irani [18]. □

Towards proving Theorem 5.1, we state definitions and a lemma. Given the input sequence $\sigma = q_1, \ldots, q_n$, let $\sigma_i = q_{j_1}, \ldots, q_{j_{n_i}}$ be the sub-sequence consisting of all queries that refer to o_i. Divide σ_i into a sequence of *groups* such that each group g_k consists of consecutive queries from σ_i and

$$\sum_{q_{j_l} \in g_k} \frac{y_{i,j_l}}{s_i} = 1. \tag{7}$$

The idea is that the cost of bypassing queries in g_k should be equal to the fetch cost f_i. If all queries are assigned to

```
SpaceEffBY (q_j)
    /*  q_j is the next query in the input sequence;
        q_j refers to object o_i and has yield y_{i,j}.
        A_obj is an algorithm for bypass-object caching.  */

    With probability y_{i,j}/s_i,
        o_i is generated as the next input for A_obj.
    The cache is maintained according to A_obj.
    If (o_i is in the cache)
        Service q_j from the cache.
    Else
        Bypass q_j to the server.
```

Figure 3. The SpaceEffBY Algorithm.

groups integrally, *i.e.*, each group either contains the whole query or no part of it, it may not be possible to satisfy Condition 7 exactly. So, when necessary, we assign a fraction of a query to one group and the rest to the next and divide the yield proportionately. A group is said to end at the last query that belongs to it. If we rearrange the queries in σ such that all queries that belong to a group are consecutive, then within a group the queries are in their original order, and the groups are ordered according to the query at which they end. We call this the *grouped* sequence, denoted by grouped(σ). All queries in σ may not be able to form a group; this happens when there aren't enough queries left for the yield to equal the object size. All such queries are dropped from grouped(σ). Let the sub-sequence of just those queries be dropped(σ). By dropping the queries in dropped(σ) from σ, we create the *trimmed* sub-sequence, denoted by trimmed(σ). This *trimmed* sequence contains queries in the same order as in σ, but some of them may be fractional. If we replace each group in grouped(σ) with the object to which the queries refer, we obtain an equivalent object sequence, denoted by object(σ). object(σ) is the very sequence sent to $\mathcal{A}_{\mathrm{obj}}$ by OnlineBY. $\mathrm{OPT}_{\mathrm{object}}$ and $\mathrm{OPT}_{\mathrm{yield}}$ are the optimal offline algorithms for bypass-object and bypass-yield caching respectively.

The following lemma states a relationship between the costs of $\mathrm{OPT}_{\mathrm{object}}$ and $\mathrm{OPT}_{\mathrm{yield}}$ in terms of object sequences and trimmed sequences.

Lemma 5.1 *Given any input sequence $\sigma = q_1, \ldots, q_n$ of queries, the cost of $\mathrm{OPT}_{\mathrm{object}}$ on object(σ) is at most 2 times the cost of $\mathrm{OPT}_{\mathrm{yield}}$ on trimmed(σ).* □

The proof of Lemma 5.1 appears in the companion technical report [26].

Among dropped(σ), let dropped$_N(\sigma)$ be the sub-sequence of dropped queries that refer to objects not in object(σ). Let trimmed$_N(\sigma)$ refer to the sub-sequence remaining when the queries in dropped$_N(\sigma)$ have been dropped from σ. Lastly, let the *cost* of dropped$_N(\sigma)$ be the sum of the cost to bypass each query in it to the server. Similarly, define cost for dropped(σ).

Observation 5.3 *The cost of* $\mathrm{OPT}_{\mathrm{yield}}$ *on* σ *is the cost of* $\mathrm{OPT}_{\mathrm{yield}}$ *on* $\mathrm{trimmed}_N(\sigma)$ *plus the cost of* $\mathrm{dropped}_N(\sigma)$.

In other words, there is no benefit in fetching objects referred to by queries in $\mathrm{dropped}_N(\sigma)$, which have a total bypass cost less than the fetch cost.

The remainder of the proof of Theorem 5.1 appears in the companion technical report. In it, we assemble the relations we have established between $\mathrm{OPT}_{\mathrm{object}}$ and $\mathrm{OPT}_{\mathrm{yield}}$ on the division of σ to complete the bound.

5.3. SpaceEffBY

SpaceEffBY is quite similar to OnlineBY (Figure 3). Instead of maintaining BYU_i values to decide when to create o_i for $\mathcal{A}_{\mathrm{obj}}$ SpaceEffBY simulates a similar effect by randomly creating o_i with probability $y_{i,j}/s_i$. The extra space taken by SpaceEffBY is $O(1)$.

6. Experiments

We develop bypass-yield caching within the SkyQuery [27] federation of astronomy databases. Caches are placed at mediators within the federation, which are the sites that receive user queries and resolve them into sub-queries for each member database. Because mediators are placed near clients, the network cost of communicating between clients and mediator sites is insignificant compared to that between clients and database servers. Thus, mediator sites act as proxy caches. We have built a prototype implementation of the above system. This allows us to evaluate the performance of the various algorithms. We base our implementation of OnlineBY and SpaceEffBY on the GDS algorithm [6, 18], because of its widespread use and known effectiveness.

Our cache is a binary heap of database objects in which heap ordering is done based on utility value. For Rate-Profile, the utility value is the *RP* value of each object. For OnlineBY, the utility value equals the BYU_i value. The heap implementation makes insertions in $O(\log k)$ time, for a heap of k objects. Evictions require $O(1)$ time. By maintaining an additional hash table on cached objects, the cache resolves hits and misses in $O(1)$ time.

We evaluate the yield of each query by re-executing the traces with the server. In case of joins and when caching columns, yields for individual objects are calculated by decomposing the yield of the entire query into component parts, corresponding to the cached objects. We demonstrate yield estimation using a typical astronomy query.

> select p.objID, p.ra, p.dec, s.z as redshift from SpecObj s, PhotoObj p where p.objID = s.objID and s.z < 0.01 and s.zConf > 0.95

When caching tables, yield for each table or view in a joined query is divided in proportion to the table's contribution to the number of attributes in the query. For the above query, yield is divided into half for each table. For attribute caching, yield is proportional to the storage of each attribute as a fraction of the total storage of all columns referenced in the query. In the above query, the total storage of all columns is 40 bytes. Storage of p.objID is 8 bytes, so its yield is $8/40 * Y$, where Y is the yield of the entire query.

Cache consistency issues do not arise with respect to the database objects. The workload has been taken from the Sloan Digital Sky Survey (SDSS), a participant in the SkyQuery federation. Once published, an SDSS database is immutable. Changes or re-calibrations of data are administered by the organization and distributed only through a new data release, *i.e.*, a new version of the database. User queries are read-only and specify a database version. However, meta-data inconsistencies might arise, especially when materialized views and indices are modified. We use the SkyQuery Web services, by which the server notifies the mediator and the cache of any changes to metadata. The cache uses this event to update metadata.

To test the effectiveness of our algorithms, we use traces gathered from the logs of the federating databases. Specifically, we use traces from two data releases – EDR and DR1 – of the largest federating node of the SkyQuery system. Each trace consists of more than 25,000 SQL requests amounting to about 1000 GB of network traffic. The SDSS traces include variety of access patterns, such as range queries, spatial searches, identity queries, and aggregate queries. Preprocessing on the traces involves removing queries that query the logs themselves.

6.1. Comparing Cache Objects

We analyze the workload traces to answer the question "What class of objects perform well in a bypass-yield cache?" and to determine the preferred object granularity. We consider query (semantic) caching versus caching database objects, such as relational tables, attributes, and materialized views. Luo *et al* [25] state that it is imperative for the workload to show both locality and containment for query caching to be viable.

The SDSS workload exhibits little query containment, which renders semantic caching ineffective. The degree of query containment is the number of queries that can be resolved from previous queries due to refinement. While determining actual query containment is NP-complete [8], an upper bound can be evaluated experimentally. Most queries in the workload look at celestial objects and their properties in a region of sky. These objects are denoted with unique identifiers. We examine queries from a continuous sub-sequence of the EDR trace in order to evaluate containment. A necessary, but not sufficient, condition for containment is that object identifiers of a subsequent query must be satisfied by object identifiers of a previous query. For clarity in presenting the data, we look at a window of 300 such queries (Figure 4). Results over larger windows are similar. Points in the chart indicate reuse of an object identi-
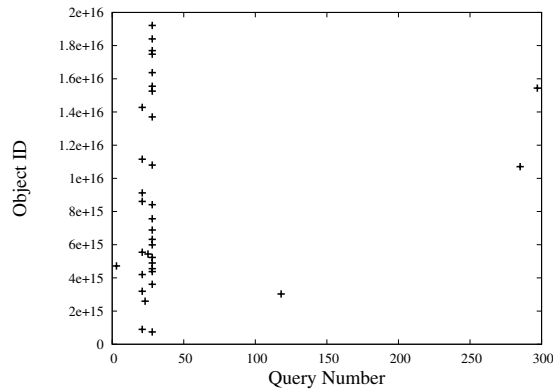
Figure 4. Query containment

Figure 5. Table locality

Figure 6. Column locality

fier in different queries, and, thus, a potential cache hit in a query cache. Our experiments indicate that few objects experience reuse in any portion of the trace over a large universe of objects. The problem is that there are few candidate celestial objects to cache, indicating few candidate queries.

Schema locality describes the reuse of (locality in) data columns and tables; the reuse of schema elements rather than specific data items. Figures 5 and 6 evaluate schema locality over the EDR trace. The x-axis corresponds to each query. On the y-axis, we enumerate columns and tables respectively. Each table is enumerated by giving a unique number between 1 and maximum number of tables in the database. Columns are enumerated by $x.y$ in which x is unique table number to which the column belongs and y is the column number in the table. Data points on the same horizontal line indicate reuse of a column or table. Both columns and tables show heavy and long lasting periods of reuse. Reuse is localized to a small fraction of the total columns or tables in the system, indicating that these columns or tables could be placed in a cache and could service many future queries as cache hits. Our algorithmic results verify this finding, showing large network traffic reductions when caching schema elements.

6.2. Performance Comparison of Algorithms

In this set of experiments, we compare the rate-based algorithm and the two on-line algorithms. We also contrast the performance of these algorithms against the base SkyQuery system (without caching) and a system that uses Greedy-Dual-Size (GDS) caching without bypass. In all experiments, we evaluate the algorithms using network cost as a metric: the total number of bytes transmitted from the databases to the proxy cache and client. Because clients and proxy caches are collocated, we do not factor traffic between them into network costs.

Rate-Profile outperforms significantly in-line (GDS) caching and SkyQuery without caching. Figures 7 and 8 show the network costs of each algorithm for columns and tables respectively. The graphs show the cumula-
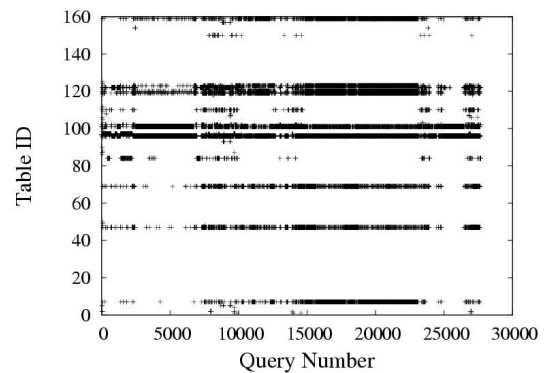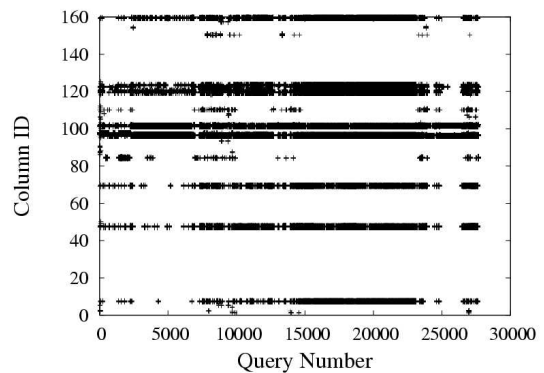
tive network usage for all queries in the trace. Bypass-yield caching reduces network load by a factor of five to twenty when compared with GDS and no caching. The "no cache" results show the sum of the size of all query results shipped from the servers. GDS performs poorly because it caches all requests, loading columns (resp. tables) into the cache and generating query results in the cache. We include results of static table caching for comparison in which a cache is populated with the optimal off-line set of tables: no cache loading or eviction occurs. While static table caching is not optimal (dynamic offline algorithms could perform better), we expect bypass-yield caches to be relatively stable when compared with caching in other models, and static table caching provides a sanity check for performance. Bypass-yield algorithms approach the performance of static table caching. Results indicate that bypass-yield algorithms realize the benefits of caching, frequent reuse and reduced network bandwidth. At the same time, they avoid the hazards of caching in database federations, preserving the data filtering benefits of evaluating queries at the servers. Bypass is the essential feature for caching the SDSS workload successfully, and the economic algorithms provide a frame-

Data Set	Number of Queries	Sequence Cost (GB)	Algorithm	Bypass Cost (GB)	Fetch Cost (GB)	Total Cost (GB)
EDR	27663	1216.94	Rate-Profile	41.08	52.84	93.92
			OnlineBY	41.06	63.38	104.44
			SpaceEffBY	83.40	42.86	126.26
DR1	24567	1980.40	Rate-Profile	126.54	75.13	201.67
			OnlineBY	130.10	68.43	198.53
			SpaceEffBY	145.28	87.35	232.63

Table 1. Cost breakdown for table caching

Data Set	Number of Queries	Sequence Cost (GB)	Algorithm	Bypass Cost (GB)	Fetch Cost (GB)	Total Cost (GB)
EDR	27663	1216.94	Rate-Profile	4.12	80.12	84.24
			OnlineBY	1.09	86.97	88.06
			SpaceEffBY	3.89	90.71	94.60
DR1	24567	1980.40	Rate-Profile	73.65	43.91	117.56
			OnlineBY	98.40	48.20	146.60
			SpaceEffBY	112.71	52.90	175.61

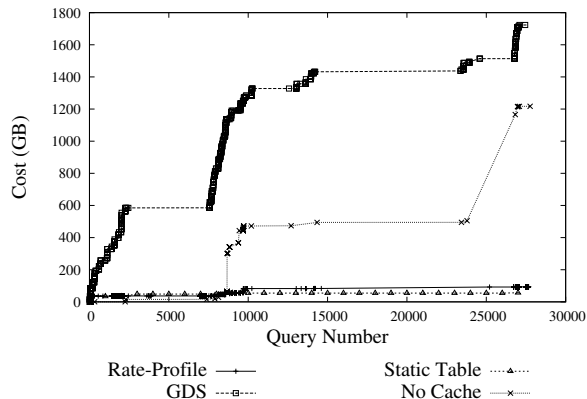**Table 2. Cost breakdown for column caching**

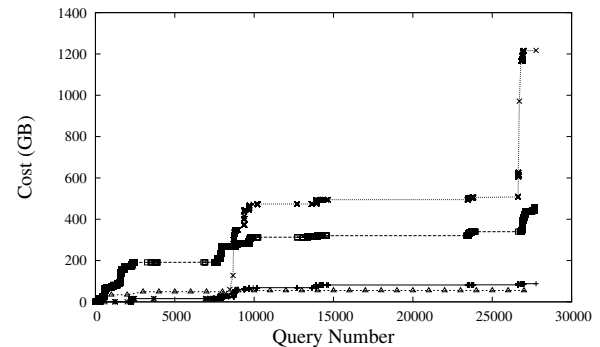**Figure 7. Network cost for table caching**

Figure 8. Network cost for column caching

work for making the bypass decision.

We also compare the performance of our three algorithms. Tables 1 and 2 show the total network costs over an entire trace and divide those costs into a bypass component, the cost of queries served at the databases, and a load component, the costs to bring objects into the cache. In most cases, Rate-Profile outperforms the on-line algorithm (OnlineBY), indicating that observed workload is a sound predictor of future access patterns. However, OnlineBY performs surprisingly well, which is promising, given that it reduces state and offers competitive bounds. The on-line randomized algorithm (SpaceEffBY) always lags behind, indicating that some amount of state aids in making the bypass decision.

The fine granularity of column caching offers benefits in all experiments when compared with table caching. In column caching, the cache is much more active; more data are fetched into the cache and fewer queries are bypassed. This results in lower overall costs when caching columns. The coarse granularity of table caching also leads to poor results for in-line (GDS) caching, which evicts and loads large tables (Figure 8).

6.3. Influence of Cache size

We examine the variability in network cost at a variety of cache sizes in order to determine the size requirements of a bypass-yield cache. Figures 9 and 10 show the performance of all algorithms as the cache size varies between 10% and 100% of the database size.
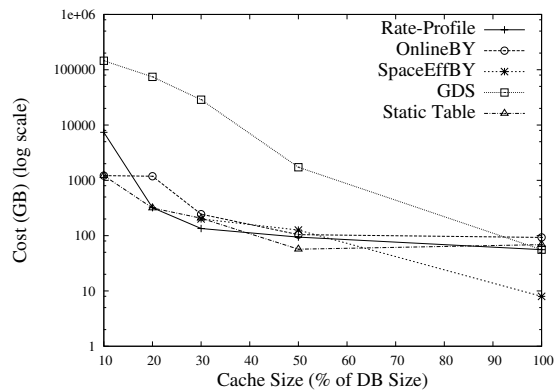
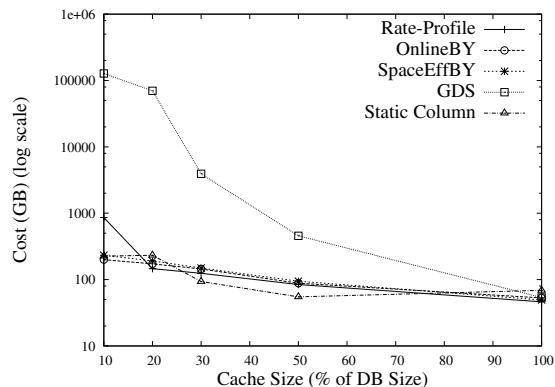**Figure 9. Performance of table caching for an increasing cache size**

Figure 10. Performance of column caching for an increasing cache size

We draw two conclusions from these results. First, the rate-based (Rate-Profile) algorithm performs poorly at very small cache sizes. The algorithm exchanges objects for those with higher rates, often evicting objects before the load cost is recovered. We expect that this artifact can be removed by tuning the algorithm. Second, bypass caches need to be relatively large, 20% to 30% of the database, to be effective. We attribute this partly to the fact that scientific databases are populated with large data items. However, we find this result to be inconclusive. The SDSS data is only about 700 MB and, thus, small relative to the amount of data in the federations we target with our technology.

Determining how the needs of cache size scale with database size remains an issue for further study. We expect that the cache size needs will not grow with database size. Rather, we expect cache size to be a function of workload. In the case of the static table caching, the load cost at small cache sizes is much more than bypass cost leading to an increase in the total cost. At 100% cache size it loads more tables that are required by the sequence.

SpaceEffBY shows a benefit of randomized algorithms in Figure 9 at a cache size of 100% of the database size. Because the algorithm loads objects randomly, with probability proportional to the yield of an individual query, the algorithm may load tables earlier or later – if at all – when compared with its deterministic relatives OnlineBY and Rate-Profile. In this case, SpaceEffBY "gets lucky" when it does not load a very large table for which future workload does not overcome the load cost. The experiment is run with a cold cache and over a finite trace, which exacerbates this effect. However, the example does show how randomized algorithms are robust to adversarial workloads.

7. Conclusions

We have presented the bypass-yield architecture for altruistic caching and network citizenship in large-scale scientific database federations. Our treatment contains several algorithms for caching within this framework, including a workload-based predictive algorithm, a competitive on-line algorithm, and a minimal-space on-line randomized algorithm. Experimental results show that all algorithms provide the benefits of caching, while preserving the filtering and parallelism benefits of database federations. Bypass-yield caching and the associated algorithms are well suited to scientific workloads, which exhibit schema locality, rather than query locality. Bypass-yield algorithms allow a cache to differentiate between queries that can be evaluated in cache to realize network savings from those that are better shipped to the servers of the federation in order to be evaluated in parallel at the data sources.

References

[1] M. Altinel, Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. G. Lindsay, H. Woo, and L. Brown. DBCache: Database caching for Web application servers. In *SIGMOD*, 2002.

[2] K. Amiri, S. Park, and R. Tewari. A self-managing data cache for edge-of-network Web applications. In *Proc. of the Conference on Information and Knowledge Management*, 2002.

[3] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for Web proxy caches. In *Proc. of the Workshop on Internet Server Performance*, 1999.

[4] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11), 2001.

[5] A. Borodin and R. El-Yaniv. *On-line Computation and Competitive Analysis.* Cambridge University Press, 1998.

[6] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. of the USENIX Symposium on Internet Technology and Systems*, 1997.

[7] B. Y. Chan, A. Si, and H. V. Leong. A framework for cache management for mobile databases: Design and evaluation. *Distributed Parallel Databases*, 10(1), 2001.

COMPUTER SOCIETY

[8] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *ACM Symposium on Theory of Computing*, 1977.

[9] L. Cherkasova and G. Ciardo. Role of aging, frequency, and size in Web cache replacement policies. In *Proc. on High Performance Computing and Networking*, 2001.

[10] E. Coffman and P. Denning. *Operating Systems Theory*. Prentice Hall, Inc, 1973.

[11] E. Cohen and H. Kaplan. LP-based analysis of Greedy-Dual-Size. In *Proc. of the ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[12] S. Dar, M. J. Franklin, B. T. Jònsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[13] H. Fujiwara and K. Iwama. Average-case competitive analyses for ski-rental problems. In *Intl. Symposium on Algorithms and Computation*, 2002.

[14] G. Gallagher. Data transport within the Distributed Oceanographic Data System. In *Proc. of the WWW Conference*, 1995.

[15] J. Gray and A. Szalay. Online science: The World-Wide Telescope as a prototype for the new computational science. Presentation at the *Supercomputing Conference*, 2003.

[16] J. M. Hellerstein. Practical predicate placement. In *SIGMOD*, 1994.

[17] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), 1988.

[18] S. Irani. Page replacement with multi-size pages and applications to Web caching. In *Proc. of the ACM Symposium on the Theory of Computing*, 1997.

[19] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *Proc. of the High-Performance Computer Architecture*. IEEE, 2003.

[20] A. Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *VLDB*, 1988.

[21] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS*, 2002.

[22] S. Jin and Z. Bestavros. Popularity-aware Greedy Dual-Size Web proxy caching. In *Proc. of the Intl. Conference on Distributed Computing Systems*, 2000.

[23] S. Jin and Z. Bestavros. Greedydual* Web caching algorithms: Exploiting two sources of temporal locality in Web request streams. *Computer Communications*, 24(2), 2001.

[24] D. Li, P. Cao, and M. Dahlin. WCIP: Web Cache Invalidation Protocol. Internet Draft, IETF, 2000.

[25] Q. Luo and J. F. Naughton. Form-based proxy caching for database-backed Web sites. In *VLDB*, 2001.

[26] T. Malik, R. Burns, and A. Chaudhary. Bypass caching: Making scientific databases good network citizens. Technical Report HSSL-2004-01, Storage Systems Lab, Johns Hopkins University, 2004.

[27] T. Malik, A. S. Szalay, A. S. Budavri, and A. R. Thakar. SkyQuery: A Web service approach to federate databases. In *Proc. of the Conference on Innovative Data Systems Research*, 2003.

[28] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. of the USENIX File and Storage Technologies Conference*, 2003.

[29] N. Niclausse, Z. Liu, and P. Nain. A new efficient caching policy for the World Wide Web. In *Proc. of the Workshop on Internet Server Performance*, 1998.

[30] E. O'Neil, P. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *ACM SIGMOD*, 1993.

[31] PlasmoDB: The plasmodium genome resource. http://www.plasmodb.org, 2002.

[32] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *VLDB*, 2000.

[33] Q. Ren and M. H. Dunham. Semantic caching and query processing. Technical report, Department of CSE, Southern Methodist University, 1998.

[34] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS. *IEEE Computer*, 19(12), 1986.

[35] P. Scheuermann, J. Shim, and R. Vingralek. Watchman: A data warehouse intelligent cache manager. In *VLDB*, 1996.

[36] http://www.sdss.org.

[37] http://www.skyquery.net.

[38] R. Stevens. *TCP/IP Illustrated Volume 1: The Protocols*. Addison-Wesley, 1994.

[39] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A new architecture for distributed data. In *ICDE*, 1994.

[40] The Times Ten Team. In-memory data management in the application tier. In *ICDE*, 2000.

[41] G. Valentin, M. Zuliani, D. Zilio, and G. Lohman. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

[42] D. Wessels and K. C. Claffy. ICP and the Squid Web cache. *IEEE Journal on Selected Areas in Communications*, 16(3), 1998.

[43] R. Wooster and M. Abrams. Proxy caching that estimates page load delays. In *Proc. of the International WWW Conference*, 1997.

[44] N. E. Young. On-line caching as cache size varies. In *Proc. of the Symposium on Discrete Algorithms*, 1991.