HAL
open science

A Dynamic Data Middleware Cache for Rapidly-Growing Scientific Repositories

Tanu Malik, Xiaodan Wang, Philip Little, Amitabh Chaudhary, Ani Thakar

To cite this version:

Tanu Malik, Xiaodan Wang, Philip Little, Amitabh Chaudhary, Ani Thakar. A Dynamic Data Middleware Cache for Rapidly-Growing Scientific Repositories. ACM/IFIP/USENIX 11th International Middleware Conference (MIDDLEWARE), Nov 2010, Bangalore, India. pp.64-84, 10.1007/978-3-642-16955-7_4 . hal-01055271

HAL Id: hal-01055271

https://inria.hal.science/hal-01055271v1

Submitted on 12 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

A Dynamic Data Middleware Cache for Rapidly-growing Scientific Repositories

Tanu Malik[1], Xiaodan Wang[2], Philip Little[3],
Amitabh Chaudhary[3], Ani Thakar[4]

[1] Cyber Center, Purdue University
tmalik@cs.purdue.edu
[2] Dept. of Computer Science, [4] Dept. of Physics and Astronomy,
Johns Hopkins University
xwang@cs.jhu.edu, thakar@pha.jhu.edu
[3] Dept. of Computer Science and Engg.,
University of Notre Dame
achaudha, plittle@cse.nd.edu

Abstract. Modern scientific repositories are growing rapidly in size. Scientists are increasingly interested in viewing the latest data as part of query results. Current scientific middleware cache systems, however, assume repositories are static. Thus, they cannot answer scientific queries with the latest data. The queries, instead, are routed to the repository until data at the cache is refreshed. In data-intensive scientific disciplines, such as astronomy, indiscriminate query routing or data refreshing often results in runaway network costs. This severely affects the performance and scalability of the repositories and makes poor use of the cache system. We present Delta a dynamic data middleware cache system for rapidly-growing scientific repositories. Delta's key component is a decision framework that adaptively *decouples* data objects—choosing to keep some data object at the cache, when they are heavily queried, and keeping some data objects at the repository, when they are heavily updated. Our algorithm profiles incoming workload to search for optimal data decoupling that reduces network costs. It leverages formal concepts from the network flow problem, and is robust to evolving scientific workloads. We evaluate the efficacy of Delta, through a prototype implementation, by running query traces collected from a real astronomy survey.
Keywords: dynamic data, middleware cache, network traffic, vertex cover, robust algorithms

1 Introduction

Data collection in science repositories is undergoing a transformation. This is remarkably seen in astronomy. Earlier surveys, such as the Sloan Digital Sky Survey (SDSS) [33, 40] collected data at an average rate of 5GB/day. The collected data was added to a database repository through an off-line process; the new repository was periodically released to users. However, recent surveys such as the Panoramic Survey Telescope & Rapid Response System (Pan-STARRS) [30] and the Large Synoptic Survey Telescope (LSST) [23] will add new data at an average rate considerably more

than 100 GB/day! Consequently, data collection pipelines have been revised to facilitate continuous addition of data to the repository [18]. Such a transformation in data collection impacts how data is made available to users when remote data middleware systems are employed.

Organizations deploy data middleware systems to improve data availability by reducing access times and network traffic [3, 34]. A critical component of middlewares systems are database caches that store subsets of repository data in close proximity to users and answer most user queries on behalf of the remote repositories. Such caches worked well with the old, batch method of data collection and release. But when data is continuously added to the repositories, cached copies of the data rapidly become stale. Serving stale data is unacceptable in sciences such as astronomy where users are increasingly interested in the latest observations. Latest observations of existing and new astronomical bodies play a fundamental role in time-domain studies and light-curve analysis [17, 32]. To keep the cached copies of the data fresh, the repository could continuously propagate updates to the cache. But this results in runaway network costs in data-intensive applications.

Indeed, transforming data middlewares to cache dynamic subsets of data for rapidly growing scientific repositories is a challenge. Scientific applications have dominant characteristics that render dynamic data caches proposed for other applications untenable. Firstly, scientific applications are data intensive. In PAN-STARRS for instance, astronomers expect daily data additions of atleast 100GB and a query traffic of 10TB each day. Consequently a primary concern is minimizing network traffic. Previously proposed dynamic data caches for commercial applications such as retail on the Web and stock market data dissemination have a primary goal of minimizing response time and minimizing network traffic is orthogonal. Such caches incorporate latest changes by either (a) invalidating subsets of cached data and then propagating updates or shipping queries, or (b) by proactively propagating updates at a fixed rate. Such mechanisms are blind to actual queries received and thus generate unacceptable amounts of network traffic.

Secondly, scientific query workloads exhibit a constant evolution in the queried data objects and the query specification; a phenomenon characteristic of the serendipitous nature of science [25, 35, 42]. The evolution often results in entirely different sets of data objects being queried in a short time period. In addition, there is no single query template that dominates the workload. Thus, it is often hard to extract a representative query workload. For an evolving workload, the challenge is in making robust decisions—that save network costs and remain profitable over a long workload sequence. Previously proposed dynamic data caches often assume a representative workload of point or range queries [5, 9, 10].

In this paper, we present Delta a dynamic data middleware cache system for rapidly growing scientific repositories. Delta addresses these challenges by incorporating two crucial design choices:

(A) Unless a query demands, no new data addition to the repository is propagated to the cache system, If a query demands the latest change, Delta first invalidates the currently available stale data at the cache. The invalidation, unlike previous systems [6, 34], is not followed by indiscriminate shipping of queries or updates; Delta incorporates

a decision framework that continually compares the cost of propagating new data additions to the cache with the cost of shipping the query to the server, and adaptively decides whether it is profitable to ship queries or to ship updates.

(B) In Delta, decisions are not made based on assuming some degree of workload stability. Often frameworks assume prior workload to be an indicator of future accesses. Such assumptions of making statistical correlation on workload patterns lead to inefficient decisions, especially in the case of scientific workloads that exhibit constant evolution.

To effectively implement the design choices, the decision framework in Delta decouples data objects; it chooses to host data objects for which it is cheaper to propagate updates, and not host data objects for which it is cheaper to ship queries. The decoupling approach naturally minimizes network traffic. If each query and update accesses a single object, the decoupling problem requires simple computation: if the cost of querying an object from the server exceeds the cost of keeping it updated at the cache, then cache it at the middleware, otherwise not. However, scientific workloads consist of SQL queries that reference multiple data objects. A general decoupling problem consists of updates and queries on multiple data objects. We show how the general decoupling problem is a combinatorial optimization problem that is NP-hard.

We develop a novel algorithm, VCover, for solving the general decoupling problem. VCover is an incremental algorithm developed over an offline algorithm for the network flow problem. VCover minimizes network traffic by profiling costs of incoming workload; it makes the best network cost optimal decisions as are available in hindsight. It is robust to changes in workload patterns as its decision making is grounded in online analysis. The algorithm also adapts well to a space constrained cache. It tracks an object's usage and makes load and eviction decisions such that at any given time the set of objects in cache satisfy the maximum number of queries from the cache. We demonstrate the advantage of VCover for the data decoupling framework in Delta by presenting Benefit, a heuristics based greedy algorithm that can also decouple data objects. Algorithms similar to Benefit are commonly employed in commercial dynamic data caches [21].

We perform a detailed experimental analysis to test the validity of the decoupling framework in real astronomy surveys. We have implemented both VCover and Benefit and experimentally evaluated their performance using more than 3 Terabyte of astronomy workload collected from the Sloan Digital Sky Survey. We also compare them against three yardsticks: NoCache, Replica and SOptimal. Our experimental results show that Delta (using VCover) reduces the traffic by nearly half even with a cache that is one-fifth the size of the server repository. Further, VCover outperforms Benefit by a factor that varies between 2-5 under different conditions. It's adaptability helps it maintain a steady performance in the scientific real-world, where queries do not follow any clear patterns.

RoadMap: We discuss related works in Section 2. The science application, *i.e.,* the definition of data objects, and the specification of queries and updates is described in Section 3. In this Section, we also describe the problem of decoupling data objects between the repository and the cache. An offline approach to the data decoupling problem is described in Section 3.1. In Section 4, we describe an online approach,

VCover, to the data decoupling problem. This approach does not make any assumptions about workload stability. An alternative approach, Benefit, which is based on workload stability is described in Section 5. We demonstrate the effectiveness of VCover over Benefit in Section 6. Finally, in Section 7 we present our conclusions and future work.

2 Related Work

The decoupling framework proposed in Delta to minimize network traffic and improve access latency is similar to the hybrid shipping model of Mariposa [38,39]. In Mariposa, processing sites either ship data to the client or ship query to the server for processing. This paradigm has also been explored in OODBMSs [11,22] More recently, applications use content distribution networks for efficiently delivering data over the Internet [31]. However, none of these systems consider propagating updates on the data [28]. In Delta the decoupling framework includes all aspects of data mobility, which include query shipping, update propagation and data object loading.

Deolasee et al. [10] consider a proxy cache for stock market data in which, adaptively, either updates to a data object are pushed by the server or are pulled by the client. Their method is limited primarily to single valued objects, such as stock prices and point queries. The tradeoff between query shipping and update propagation are explored in online view materialization systems [20, 21]. The primary focus is on minimizing response time to satisfy currency of queries. In most systems an unlimited cache size is assumed. To compare, in Benefit we have developed an algorithm that uses workload heuristics similar to the algorithm developed in [21]. The algorithm minimizes network traffic instead of response time. Experiments show that such algorithms perform poorly on scientific workloads.

More recent work [16] has focused on minimizing the network traffic. However the problem is focused on communicating just the current value of a single object. As a result the proposed algorithms do not scale for scientific repositories in which objects have multiple values. Alternatively, Olston et al. [27, 29] consider the precision-based approach to reduce network costs. In their approach, users specify precision requirements for each query, instead of currency requirements or tolerance for staleness. In scientific applications such as the SDSS, users have zero tolerance for approximate or incorrect values for real attributes as an imprecise result directly impacts scientific accuracy.

Finally, there are several dynamic data caches that maintain currency of cached data. These include DBProxy [1], DBCache [3], MTCache [14] and TimesTen [41] and the more recent [13]. These systems provide a comprehensive dynamic data caching infrastructure that includes mechanisms for shipping queries and updates, defining data object granularity and specifying currency constraints. However, they lack a decision framework that adaptively chooses between query shipping and update propagation and data loading; any of the data communication method is deemed sufficient for query currency.

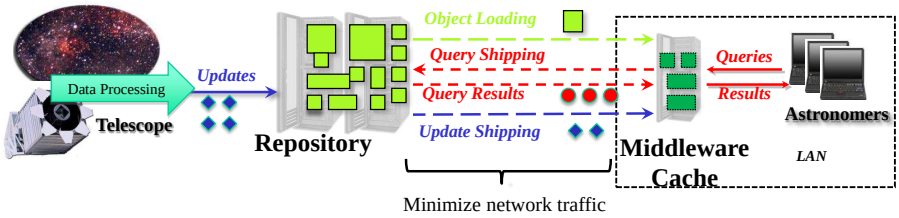

Fig. 1. The Delta architecture.

3 Delta: A Dynamic Data Middleware Cache

We describe the architectural components in Delta and how data is exchanged between the server repository and the cache (See Figure 1). Data of a scientific repository is stored in a relational database system. While the database provides a natural partition of the data in the form of tables and columns, often spatial indices are available that further partition the data into "objects" of different sizes. A rapidly-growing repository receives updates on these data objects from a data pipeline . The updates, predominantly, insert data into, and in some cases, modify existing data objects. Data is never deleted due to archival reasons. We model a repository as a set of data objects $S = o_1, \ldots, o_N$. Each incoming update u affects just one object $o(u)$, as is common in scientific repositories.

Data objects are stored at the cache of the middleware system to improve data availability and reduce network traffic. The cache is located along with or close to the clients, and thus "far" from the repository. The cache has often much less capacity than the original server repository and is thus space-constrained. We model the cache again as a subset of data objects $C = o_1, \ldots, o_n$. The objects are cached in entirety or no part of it is cached. This simplifies loading of objects. Objects at the cache are invalidated when updates arrive for them at the server. Each user query, q, is a read-only SQL-like query that accesses data from a set of data objects $B(q)$. The cache answers some queries on the repository's behalf. Queries that cannot be answered by the cache are routed to the repository and answered directly from there. To quantify its need for latest data, queries may include user or system specified currency requirements in form of a *tolerance for staleness* $t(q)$, defined as follows: Given $t(q)$, an answer to q must incorporate all updates received on each object in $B(q)$ except those that arrived within the last $t(q)$ time units. This is similar to the syntax for specifying $t(q)$ as described in [14]. The lower the tolerance, the stricter is the user's need for current data.

To satisfy queries as per their tolerance for staleness, the cache chooses between the three available communication mechanisms: (a) *update shipping*, (b) *query shipping*, and (c) *object loading*. To ship updates, the system sends an update specification including any data for insertion and modification to be applied on the cached data objects. In shipping queries, the system redirects the query to the repository. The up-to-date result is then sent directly to the client. Through the object loading mechanism, the cache loads objects not previously in the cache, provided there is available space to load. The three mechanisms differ from each other in terms of semantics and the cost of using

them. For instance, in update shipping only the newly inserted tuples to an object are shipped whereas in object loading the entire data object (including the updates) is shipped.

The system records the the cost of using each of the data communication mechanism. In Delta, we have currently focused on the network traffic costs due to the use of these mechanisms. Latency costs are discussed in Section 4. Network traffic costs are assumed proportional to the size of the data being communicated. Thus when an object o is loaded, a load cost, $\nu(o)$, proportional to the object's size is added to the total network traffic cost. For each query q or update u shipped a network cost $\nu(q)$ or $\nu(u)$ proportional to the size of q's result or the size of data content in u is added respectively. The proportional assumption 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 [37]. The quantitative difference in costs between the three mechanisms is described through an example in Section 3.1.

In Delta the difference, in terms of cost, of using each communication mechanism leads to the formulation of the data decoupling problem. We first describe the problem in words and then for ease of presentation also provide a graph-based formulation.

The data decoupling problem: In the data decoupling problem we are given the set of objects on the repository, the online sequence of user queries at the cache, and the online sequence of updates at the repository. The problem is to decide which objects to load into the cache from the repository, which objects to evict from the cache, which queries to ship to the repository from the cache, and which updates to ship from the repository to the cache such that (a) the objects in cache never exceed the cache size, (b) each query is answered as per its currency requirement, and (c) the total costs, described next, are minimized. The total costs are the sum of the load costs for each object loaded, the shipping costs for each query shipped, and the shipping costs for each update shipped.

Graph-based formulation: The decoupling problem given a set of objects in the cache, can be visualized through a graph $G(V, E)$ in which the vertices of the graph are queries, updates and data objects. Data object vertices are of two kinds: one that are in cache, and ones that are not in cache. In the graph, an edge is drawn between (a) a query vertex and an update vertex, if the update affects the query, and (b) a query vertex and a data object vertex, if the object is not in cache and is accessed by the query. In the graph, for presentation, edges between an update vertex and an object not in cache and query vertex and an object in cache are not drawn. Such a graph construction helps to capture the relationships between queries, updates and objects. We term such a graph as an *interaction* graph as the edges involve a mutual decision as to which data communication mechanism be used. The decoupling problem then corresponds to determining which data communication to use since the right combination minimizes network traffic costs.

The interaction graph is fundamental to the data decoupling problem as algorithms can determine an optimal decoupling or the right combination of data communication mechanisms to use. The algorithms must also be robust to incoming query and update workloads since slight changes in the workload lead to entirely different data communication mechanisms becoming optimal (See design choices 1 and 1 in Section 1). This is

demonstrated in the next subsection through an example. The example demonstrates that different choices become optimal when the entire sequence of queries, updates and their accompanying costs are known in advance, and there are no cache size constraints.

3.1 Determining off-line, optimal choices

The objective of an algorithm for a data decoupling problem is to make decisions about which queries to ship, which updates to ship, and which objects to load such that the cost of incurred network traffic is minimized. This is based on an incoming workload pattern. The difficulty in determining the optimal combination of decision choices arises because a slight variation in the workload, *i.e.,* change in cost of shipping queries or shipping updates or the query currency threshold, results in an entirely different combination being optimal. This is best illustrated through an example.

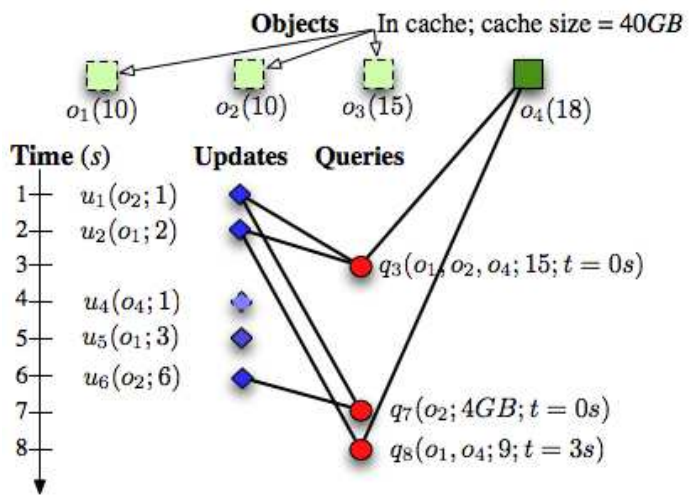

Fig. 2. An decoupling graph for a sample sequence. The notation: $o_1(10)$ implies size of object o_1 is 10 GB, which is also its network traffic load cost. $u_1(o_2, 1)$ implies update u_1 is for object o_2 and has a network traffic shipping cost of $1GB$. $q_3(o_1, o_2, o_4; 15; t = 0s)$ implies query q_3 accesses objects o_1, o_2, and o_4; has a network traffic shipping cost of $15GB$; and has no tolerance for staleness.

In the Figure2, among the four data objects o_1, o_2, o_3, o_4, objects o_1, o_2, o_3 have been loaded into the cache (broken lines) and object o_4 is currently not in cache (solid lines), and will be available for loading if there is space. Consider a sequence of updates and queries over the next eight seconds. Based on the graph formulation of the decoupling problem, we draw edges between queries, updates and the objects. Query q_3 accesses the set o_1, o_2, o_4 and has edges with o_4, u_1, and u_2. since either q_3 is shipped or the use of other data communication mechanism is necessary, namely o_4 is loaded, and u_1 and u_2 are shipped. For q_7, since object o_2 is in cache, the dependence is only on

updates u_1 and u_6 both of which are necessary to satisfy q_7's currency constraints. q_8 accesses set o_1, o_4 and so has edges with o_4 and u_2. Note, it does not have edges with u_4 because shipping u_4 is not necessary till object o_4 is loaded into the cache. If o_4 was loaded before u_4 arrives, then an edge between u_4 and q_8 will be added. There is no edge with u_5 since the tolerance for staleness of q_8 permits an answer from the cached copy of o_1 that need not be updated with u_5.

In the above example, the right choice of actions is to evict o_3 and load o_4 at the very beginning. Then at appropriate steps ship updates u_1, u_2, and u_4, and the query q_7. This will satisfy all currency requirements, and incur a network traffic cost of $26GB$. Crucial to this being the best decision is that q_8's tolerance for staleness allows us to omit shipping u_5. If that were not the case, then an entirely different set of choices become optimal which include not loading o_4, and just shipping queries q_3, q_7, and q_8 for a network cost of $28GB$. Such a combination minimizes network traffic.

In the above example, if we focus only on the internal interaction graph of cached objects, which is formed with nodes u_1, u_6 and q_7, we observe that determining an optimal decision choice corresponds to finding the minimum-weight vertex cover [8, 43] on this subgraph. For lack of space, we omit a description on minimum-weight vertex cover and directly state the correspondence through the following theorem:

Theorem 1 (Min Weight Vertex Cover). *Let the entire incoming sequence of queries and updates in the internal interaction graph G be known in advance. Let VC be the minimum-weight vertex cover for G. The optimal choice is to ship the queries and the updates whose corresponding nodes are in VC.*

Proof Sketch. If the entire incoming sequence of queries and updates is known we claim the optimal choice is: (1) choose such that every query q is either shipped or all updates interacting with q are shipped, and (2) make a choice such that the total weights of nodes chosen for shipping is minimized. This is true since the sum of the weights of nodes chosen is equal to the actual network traffic cost incurred for the choice. This corresponds to the minimum-weight vertex cover problem (see definition in [12]). □

The minimum-weight vertex cover is NP-hard problem in general [36]. However, for our specific case the sub-graph is a bi-partite graph in that no edges exist amongst the set of query nodes and the set of update nodes, but only between query and update nodes. Thus we can still solve the minimum-weight vertex cover problem by reducing it to the *maximum network flow* problem [15]. A polynomial time algorithm for the maximum network flow problem is the Edmonds-Karp algorithm [8]. This algorithm is based on the fact that a flow is maximum if and only if there is no augmenting path. The algorithm repeatedly finds an augmenting path and augments it with more flow, until no augmenting path exists.

A primary challenges in employing an algorithm for the maximum network flow problem to Delta is in (a) knowing the sequence of events in advance, and (b) extending it a limited size cache. The computation of the max-cover (or equivalently determining the max flow) changes as different knowledge of the future becomes available. In the example, if we just did not know what would happen at time $8s$, but knew everything before, we would not load o_4, and instead ship q_3 and q_7. Then when q_8 arrives, we

would ship it too. Thus, different partial knowledge of the future can lead to very different decisions and costs. The decisions obtained by the computation of the max-cover applies only to the objects in cache. We still need a mechanism for profitably loading objects in the cache.

4 VCover

VCover is an online algorithm for the data decoupling problem. It determines objects for which queries must be shipped and object for which updates must be shipped. The algorithm makes decisions in an "online" fashion, using minimal information about the incoming workload. In VCover we also address the second challenge of a space-constrained cache. The overall algorithm is shown in Figure 3.

The algorithm relies on two internal modules UpdateManager and LoadManager to make two decisions: When a query arrives, VCover first determines if the all objects accessed by the query are in cache. If all objects are in cache, this query is presented to the UpdateManager which chooses between shipping the outstanding updates required by the query, or shipping the query itself. If, instead, a query arrives that accesses at least one object not in cache, then VCover ships the query to the server, and also presents it to the LoadManager which decides, in background, whether to load the missing object(s) or not. We now describe the algorithms behind the UpdateManager and the LoadManager.

```
VCover
Invocation: By arriving query q, accessing objects B(q), with network traffic cost
  ν(q).
Objective: To choose between invoking UpdateManager, or shipping q and invoking
  LoadManager.
if all objects in B(q) are in cache then
    UpdateManager with query q
else
    Ship q to server. Forward result to client
    In background: LoadManager with query q
```

Fig. 3. The main function in VCover.

UpdateManager The UpdateManager builds upon the offline framework presented in Section 3.1 to make "online" decisions as queries and updates arrive. The UpdateManager incorporates the new query into an existing internal interaction graph G' by adding nodes for it and the updates it interacts with and establishes the corresponding edges. To compute the minimum-weight vertex cover it uses the the Edmonds-Karp algorithm, which finds the maximum network flow in a graph. However, instead of

running a network flow algorithm each time a query is serviced, it uses an incremental algorithm that finds just the change in flow since the previous computation (Figure 4). If the result of the vertex cover computation is that the query should be shipped, then the UpdateManager does accordingly (Lines 21–23). If not, then some updates are shipped.

```
9  UpdateManager on cache
10 Invocation: By VCover with query q accessing objects B(q), with network traffic cost
   ν(q).
11 Objective: To choose between shipping q or shipping all its outstanding interacting
   updates, and to update the interaction graph.
12 if each update interacting with q has been shipped then
13     Execute q in cache. Send result to client
14 Add query vertex q with weight (ν(q)) to existing internal interaction graph G' to get G
15 foreach object o ∈ B(q) marked stale do
16     foreach outstanding update u for o do
17         Add update vertex u to G, with weight its shipping cost, if not already present
18 foreach update vertex u interacting with q do
19     Add edge (u, q) to G
20 Compute minimum-weight vertex cover VC of G using incremental network-flow
   algorithm
21 if q ∈ VC then
22     if q not already executed on cache then
23         Ship q to server. Forward result to client
```

Fig. 4. UpdateManager on cache.

The key observation in computing the incremental max-flow is that as vertices and edges are added and deleted from the graph there is no change in the previous flow computation. Thus previous flow remains a valid flow though it may not be maximum any more. The algorithm therefore begins with a previous flow and searches for augmenting paths that would lead to the maximum flow. As a result, for any sequence of queries and updates, the total time spent in flow computation is not more than that for a single flow computation for the network corresponding to the entire sequence. If this network has n nodes and m edges, this time is $O(nm^2)$ [8]—much less than the $O(n^2m^2)$-time the non-incremental version takes.

The UpdateManager does not record the entire sequence of queries and updates to make decisions. The algorithm always works on a *remainder* subgraph instead of the subgraph made over all the queries and updates seen so far. This remainder subgraph is formed from an existing subgraph by excluding all update nodes picked in a vertex cover at any point, and all query nodes not picked in the vertex cover. The exclusion can be safely done because in selecting the cover at any point the shipping of an

1. Let G' be the previous internal interaction graph and H' the corresponding network constructed in previous iteration.
2. Let G be the current internal interaction graph.
3. Add source and sink vertices to G and corresponding capacitated edges as described in [15] to construct network H.
4. Find maximum flow in H, based on the known maximum flow for H'.
5. Determine minimum-weight vertex for G from maximum flow values of H, again as described in [15].

Fig. 5. Single iteration of the incremental network flow algorithm.

update is justified based only on the past queries it interacts with, and not with any future queries or updates and therefore will never be part of future cover selection. This makes computing the cover robust to changes in workload. This technique also drastically reduces the size of the working subgraph making the algorithm very efficient in practice.

Managing Loads VCover invokes the LoadManager to determine if it is useful to load objects in the cache and save on network costs or ship the corresponding queries. The decision is only made for those incoming queries which access one or more objects not in cache. The difficulty in making this decision is due to queries accessing multiple objects, each of which contributes a varying fraction to the query's total cost, $\nu(q)$. The objective is still to find, in an online fashion, the right combination of objects that should reside in cache such that the total network costs are minimized for these queries.

The algorithm in LoadManager builds upon the popular Greedy-Dual-Size (GDS) algorithm [7] for Web-style proxy caching. GDS is an object caching algorithm that calculates an object's usage in the cache based on its frequency of usage, the cost of downloading it and its size. If the object is not being heavily used, GDS evicts the object. In GDS an object is loaded as soon it is requested. Such a loading policy can cause too much network traffic. In [24], it is shown that to minimize network traffic, it is important to ship queries on an object to the server until shipping costs equal to the object load costs have been incurred. After that any request to the object must load the object into the cache. The object's usage in the cache is measured from frequency and recency of use and thus eviction decisions can be made similar to GDS, based on object's usage in cache.

In VCover queries access multiple objects and the LoadManager uses a simple twist on the algorithm in [24] that still ensures finding the right combination of choices. The LoadManager (a) randomly assigns query shipping costs among objects that a query accesses, and (b) defers to GDS to calculate object's usage in the cache. The random assignment ensures that in expectation, objects are made *candidates* for load only when cost attributed equals the load cost. A candidate for load is considered by considering in random sequence the set of objects accessed by the query (Line 29 in Figure 6). At this point, the shipping cost could be attributed to the object.

The load manager employs two techniques that makes its use of Greedy-Dual-Size more efficient. We explain the need for the techniques:

- Explicit tracking of the total cost (due to all queries) of each object is inefficient. The load manager eliminates explicit tracking of the total cost by using randomized loading (Lines 27-35). This leads to a more space-efficient implementation of the algorithm in which counters on each object are not maintained. When the shipping cost from a single query covers the entire load cost of an object, the object is immediately made a candidate to be loaded. Else, it is made so with probability equal to the ratio of the cost attributed from the query and the object's load cost.
- In a given subsequence of load requests generated by q, say $o_1, o_2, \ldots, o_m$, it is possible that the given $\mathcal{A}_{\text{obj}}$ loads an o_i only to evict it to accommodate some later o_j in the *same* subsequence. Clearly, loading o_i is not useful for the LoadManager. To iron out such inefficiencies we use a lazy version of $\mathcal{A}_{\text{obj}}$, in our case Greedy-Dual-Size.

In the LoadManager once the object is loaded, the system ensures that all updates that came while the object was being loaded are applied and the object is marked fresh by both cache and server.

```
24 LoadManager on cache
25 Invocation: By VCover with query q, accessing objects B(q), with network traffic cost
   ν(q).
26 Objective: To load useful objects
27 c ← ν(q)
28 while B(q) has objects not in cache and c > 0 do
29     o ← some object in B(q) not in cache
30     if c ≥ l(o) then                                  /* l(o) is load cost of o */
31         A_obj_lazy with input o
32         c ← c − l(o)
33     else                                              /* randomized loading */
34         With probability c/l(o): A_obj_lazy with input o
35         c ← 0
36     Load and evict objects according to A_obj_lazy
37     if A_obj_lazy loads o then
38         Both server and cache mark o fresh
```

Fig. 6. LoadManager on cache given an object caching algorithm $\mathcal{A}_{\text{obj}}$.

Discussion: There are several aspects of VCover that we would like to highlight. First, in Delta we have focused on reducing network traffic. Consequently, in presenting VCover we have included only those decisions that reduce network traffic. These decisions naturally decrease response times of queries that access objects in cache. But queries for which updates need to be applied may be delayed. In some applications, such as weather

prediction, which have similar rapidly-growing repositories, minimizing overall response time is equally important. To improve the response time performance of delayed queries, some updates can be *preshipped*, *i.e.,* proactively sent by the server. For lack of space we have omitted how preshipping in VCover can further improve overall response times of all queries. We direct the reader to the accompanying technical report [26] for this.

In the Delta architecture data updates correspond predominantly to data inserts (Section 3). This is true of scientific repositories. However the decision making in VCover is independent of an update specification and can imply any of the data modification statements *viz.* insert, delete or modify. Thus we have chosen the use of the term update.

LoadManager adopts a randomized mechanism for loading objects. This is space-efficient as it obviates maintaining counters on each object. This efficiency is motivated by meta-data issues in large-scale remote data access middlewares that cache data from several sites [26].

Finally, an implementation of VCover requires a semantic framework that determines the mapping between the query, q, and the data objects, $B(q)$, it accesses. The complexity of such a framework depends upon the granularity at which data objects are defined. If the objects are tables or columns, such a mapping can be determined by the specification of the query itself. If the objects are tuples, finding such a mapping a-priori is difficult. However for most applications such a mapping can be found by exploiting the semantics. For instance, in astronomy, queries specify a spatial region and objects are also spatially partitioned. Thus finding a mapping can be done by some pre-processing as described in Section 6.

5 Benefit: An alternative approach to the decoupling problem

In VCover we have presented an algorithm that exploits the combinatorial structure (computing the cover) within the data decoupling problem and makes adaptive decisions using ideas borrowed from online algorithms [4]. An alternative approach to solving the data decoupling problem is an exponential smoothing-based algorithm that makes decisions based on heuristics. We term such an algorithm Benefit as it is inherently greedy in its decision making.

In Benefit we divide the sequence of queries and updates into windows of size δ. At the beginning of each new window i, for each object currently in the cache, we compute the "benefit" b_{i-1} accrued from keeping it in the cache during the past window $(i-1)$. b_{i-1} is defined as the network cost the object saves by answering queries at the cache, less the amount of traffic it causes by having updates shipped for it from the server. Since a query answered at the cache may access multiple objects, the cost of its shipping (which is saved) is divided among the objects the query accesses in proportion to their sizes. This form of dividing the cost has been found useful in other caches as well [2,24].

For each object currently not in cache, we compute similarly the benefit it would have accrued if it *had been* in the cache during window $(i-1)$. But here, we further reduce the benefit by the cost to load the object.

A forecast μ_i of the benefit an object will accrue during the next window i is then computed using exponential smoothing: $\mu_i = (1-\alpha)\mu_{i-1}+\alpha b_{i-1}$, in which μ_{i-1} was the forecast for the previous window, and α, which is $0 \leq \alpha \leq 1$, is a learning parameter.

We next consider only objects with positive μ_i values and rank them in decreasing order. For window i, we greedily load objects in this order, until the cache is full. Objects which were already present in cache in window $(i-1)$ don't have to be reloaded. For lack of space, pseudo-code of Benefit is not presented in this paper, but is included in the accompanying technical report.

Benefit reliance on heuristics is similar to previously proposed algorithms for online view materialization [20,21]. It, however, suffers from some weaknesses. The foremost is that Benefit ignores the combinatorial structure within the problem by dividing the cost of shipping a query among the objects the query accesses in proportion to their sizes. This difference is significant in that Benefit's performance can be proved analytically. However, in VCover the combinatorial structure leads to a mathematical bound on its performance. For details on the proof of performance, the reader is directed to the technical report [26]. Further, Benefit's decision making is heavily dependent on the size of window chosen. It also needs to maintain state for each object (the μ_i values, the queries which access it, etc.) in the database irrespective of whether it is in the cache or not.

6 Empirical Evaluation

In this section, we present an empirical evaluation of Delta on real astronomy query workloads and data. The experiments validate using a data decoupling framework for minimizing network traffic. Our experimental results show that Delta (using VCover) reduces the traffic by nearly half even with a cache that is one-fifth the size of the server. VCover outperforms Benefit by a factor that varies between 2-5 under different conditions.

6.1 Experimental Setup

Choice of Survey: Data from rapidly growing repositories, such as those of the Pan-STARRS and the LSST, is currently unavailable for public use. Thus we used the data and query traces from the SDSS for experimentation. SDSS periodically publishes updates via new data release. To build a rapidly growing SDSS repository, we simulated an update trace for SDSS in consultation with astronomers. The use of SDSS data in validating Delta is a reasonable choice as the Pan-STARRS and the LSST databases have a star-schema similar to the SDSS. When open to public use, it is estimated that the Pan-STARRS and the LSST repositories will be queried similar to the SDSS.

Setup: Our prototype system consists of a server database and a middleware cache database, both implemented on an IBM workstation with 1.3GHz Pentium III processor and 1GB of memory, running Microsoft Windows, each running a MS SQL Server 2000 database. A sequence of data updates are applied to a server database Queries, each with a currency specification criteria, arrive concurrently at a cache database. To satisfy queries with the latest data, the server and cache database uses MS SQL

Server's replica management system to ship queries and updates, and, when necessary, it uses bulk copying for object loading. The decisions of when to use each of the data communication mechanism is dictated by the optimization framework of Delta, implemented as stored procedures on the server and cache databases.

Server Data: The server is an SDSS database partitioned in spatial data objects. To build spatial data objects, we use the primary table in SDSS, called the PhotoObj table, which stores data about each astronomical body including its spatial location and about 700 other physical attributes. The size of the table is roughly 1TB. The table is partitioned using a recursively-defined quad tree-like index, called the *hierarchical triangular mesh* [19]. The `HTM` index conceptually divides the sky into partitions; in the database these partitions translate into roughly equi-area data objects. A partitioning of the sky and therefore the data depends upon the level of the HTM index chosen. For most experiments we used a level that consisted of 68 partitions (ignoring some which weren't queried at all), containing about 800 GB of data. These partitions were given object-IDs from 1 to 68. Our choice of 68 objects is based on a cache granularity experiment explained in Section 6.2. The The data in each object varies from as low as 50 MB to as high as 90 GB.

Growth in Data: Updates are in the form of new data inserts and are applied to a spatially defined data object. We simulated the expected update patterns of the newer astronomy surveys under consultation with astronomers. Telescopes collect data by scanning specific regions of the sky, along great circles, in a coordinated and systematic fashion [40]. Updates are thus clustered by regions on the sky. Based on this pattern, we created a workload of 250,000 updates. The size of an update is proportional to the density of the data object.

Queries: We extracted a query workload of about 250,000 queries received from January to February, 2009. The workload trace consists of several kinds of queries, including range queries, spatial self-join queries, simple selection queries, as well as aggregation queries. Preprocessing on the traces involves removing queries that query the logs themselves. In this trace, 98% of the queries and 99% of the network traffic due to the queries is due to the PhotoObj table or views defined on PhotoObj. Any query which does not query the PhotoObj is bypassed and shipped to the server.

Costs: The traffic cost of shipping a query is the actual number of bytes in its results on the current SDSS database. The traffic cost of shipping an update was chosen to match the expected 100 GB of update traffic each day in the newer databases.

A sample of the query and update event sequence is shown in Figure 7. The sequence is along the x-axis, and for each event, if it is an update, we put a blue diamond next to the object-ID affected by the update. If the event is a query, we put a yellow dot next to *all* object-IDs accessed by the query. Even though this rough figure only shows a sample of the updates and queries, and does not include the corresponding costs of shipping, it supports what we discovered in our experiments: object-IDs 22, 23, 24, 62, 63, 64 are some of the query hotspots, while object-IDs 11, 12, 13, 30, 31, 32 are some of the update hotspots. The figure also indicates that real-world queries do not follow any clear patterns.

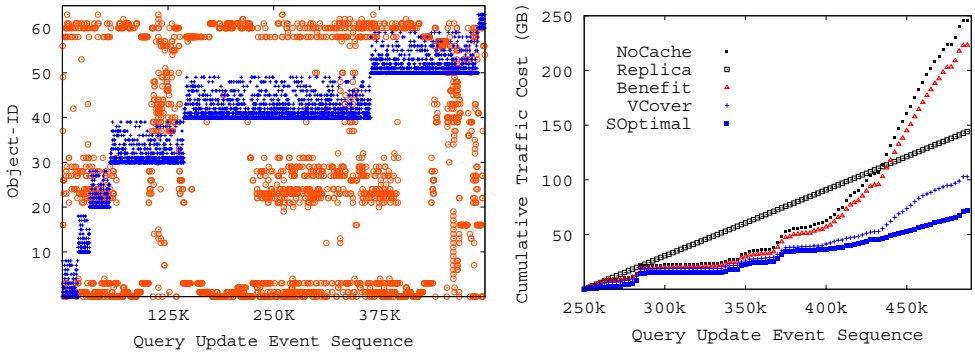

Fig. 7. (a) Object-IDs corresponding to each query (red ring) or update (blue cross) event. Queries evolve and cluster around different objects over time. (b) Cumulative traffic cost. VCover, almost as good as SOptimal, outperforms others.

Two Algorithms, Three Yardsticks We compare the performances of VCover, the core algorithm in Delta with Benefit, the heuristic algorithm that forecasts using exponential smoothing. We also compare the performance of these two algorithms with three other policies. These policies act as yardsticks in that the algorithm can be considered poor or excellent if it performs better or worse than these policies. The policies are:

- **NoCache**: Do not use a cache. Ship all queries to the server. Any algorithm (in our case VCover and Benefit), which has a performance worse than NoCache is clearly of no use.
- **Replica**: Let the cache be as large as the server and contain all the server data. To satisfy queries with the most current data, ship all updates to the cache as soon as they arrive at server. If VCover and Benefit, both of which respect a cache size limitation, perform better than Replica they are clearly good.
- **SOptimal**: Decide on the best *static set of objects* to cache *after* seeing the entire query and update sequence. Conceptually, its decision is equivalent to the single decision of Benefit using a window-size as large as the entire sequence, but in an offline manner. To implement the algorithm we loads all objects it needs at the beginning and do not ever evict any object. As updates arrive for objects in cache they are shipped. Any online algorithm, which cannot see the queries and updates in advance, but with performance close to the SOptimal is outstanding.

Default parameter values, warmup period Unless specified otherwise, in the following experiments we set the cache size to 30% of server size,and the window size δ in Benefit to 1000. The choice are obtained by varying the parameters in the experiment to obtain the optimal value. The cache undergoes an initial warm-up period of about 250,000 events for both VCover and Benefit. A large warm-up period is a characteristic of this particular workload trace in which queries with small query cost occur earlier in trace. As a result objects have very low probability of load. In this warm-up period the cache remains nearly empty and almost all queries are shipped. In general, our experience with other workload traces of similar size have shown that a range of a

warm period can be anywhere from 150,000 events to 300,000 events. To focus on the more interesting post warm-up period we do not show the events and the costs incurred during the warmup period.

6.2 Results

Minimizing traffic cost The cumulative network traffic cost along the query-update event sequence for the two algorithms and three yardsticks is in Figure 7(a). VCover is clearly superior to NoCache and Benefit: as more query and update events arrive it continues to perform better than them and at the end of the trace has an improvement by a factor of at least 2 with Benefit, and to Replica by a factor about 1.5. (For replica load costs and cache size constraints are ignored.) Further, as more data intensive queries arrive, Benefit is barely better than NoCache. VCover closely follows SOptimal until about Event 430K, when it diverges, leading to a final cost about 40% (35 GB) higher. On closer examination of the choices made by the algorithm, we discovered that, with hindsight, SOptimal determines that Object-ID 39 of 38 GB and Object-ID 29 of 4 GB would be useful to cache and loads them at the beginning. But VCover discovers it only after some high-shipping cost queries that arrive for these objects, and loads them around Event 430K, thus paying both the query shipping cost and load cost.

Varying number of updates A general-purpose caching algorithm for dynamic data should maintain its performance in the face of different rates of updates and queries. In Figure 8(b) we plot the final traffic cost for each algorithm for different workloads, each workload with the same 250,000 queries but with a different number of updates. The simplistic yardstick algorithms NoCache and Replica do not take into account the relative number of updates and queries. Since the queries remain the same, the cost of NoCache is steady at 300 GB. But as the number of updates increase the cost of Replica goes up: the three-fold increase in number of updates results in a three-fold increase in Replica's cost. The other three algorithms, in contrast, show only a slight increase in their cost as the number of updates increase. They work by choosing the appropriate objects to cache for each workload, and when the updates are more, they compensate by keeping fewer objects in the cache. The slight increase in cost is due to query shipping cost paid for objects which are no longer viable to cache. This experiment illustrates well the benefits of Delta irrespective of the relative number of queries and updates.

Choice of objects In Figure 8(a) we plot VCover's cumulative traffic cost, along the event sequence, for different choices of data objects. Each choice corresponds to a different level in the quad tree structure in SDSS, from a set of 10 objects corresponding to large-area upper-level regions, to a set of 532 objects corresponding to small-area lower-level regions. Each set covers the entire sky and contains the entire data. The performance of VCover improves dramatically as number of number of objects increase (sizes reduce) until it reaches 91 and then begins to slightly worsen again. The initial improvement is because as objects become smaller in size, less space in the cache is wasted, the hotspot decoupling is at a finer grain, and thus more effective. But this trend reverses as the objects become too small since the likelihood that future queries are entirely contained in the objects in cache reduces, as explained next. It has been observed in scientific databases that it is more likely that future queries access data which is "close to" or "related to," rather than the exact same as, the data accessed

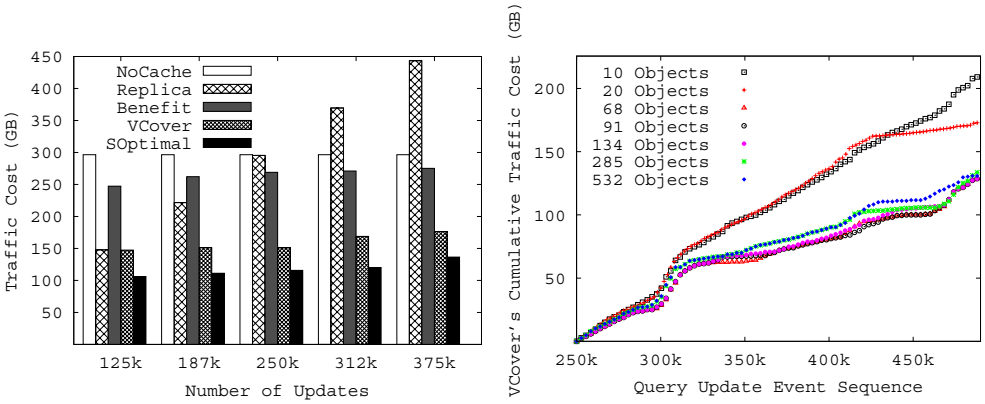

Fig. 8. (a) Traffic cost for varying number of updates. (b) VCover's cumulative traffic cost for different choices of object sets.

by current queries [24]. In astronomy, e.g., this is partly because of several tasks that scan the entire sky through consecutive queries.

7 Conclusion

Repositories in data-intensive science are growing rapidly in size. Scientists are also increasingly interested in the time dimension and thus demand latest data to be part of query results. Current caches provide minimal support for incorporating the latest data at the repository; many of them assume repositories are static. This often results in runaway network costs.

In this paper we presented Delta a dynamic data middleware cache system for rapidly growing repositories. Delta is based on a data decoupling framework—it separates objects that are rapidly growing from objects that are heavily queried. By effective decoupling the framework naturally minimizes network costs. Delta relies on VCover, a robust, adaptive algorithm that decouples data objects by examining the cost of usage and currency requirements. VCover's decoupling is based on sound graph theoretical principles making the solution nearly optimal over evolving scientific workloads. We compare the performance of VCover with Benefit, a greedy heuristic, which is commonly employed in dynamic data caches for commercial applications. Experiments show that Benefit scales poorly than VCover on scientific workloads in all respects.

A real-world deployment of Delta would need to also consider several other issues such as reliability, failure-recovery, and communication protocols. The future of applications, such as the Pan-STARRS and the LSST depends on scalable network-aware solutions that facilitate access to data to a large number of users in the presence of overloaded networks. Delta is a step towards meeting that challenge.

Acknowledgments The authors sincerely thank Alex Szalay for motivating this problem to us, and Jim Heasley for describing network management issues in designing

database repositories for the Pan-STARRS survey. Ani Thakar acknowledges support from the Pan-STARRS project.

References

1. K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: a dynamic data cache for web applications. In *Proc. Int'l. Conf. on Data Engineering*, 2003.
2. A. Bagchi, A. Chaudhary, M. T. Goodrich, C. Li, and M. Shmueli-Scheuer. Achieving communication efficiency through push-pull partitioning of semantic spaces to disseminate dynamic information. *Transactions on Knowledge and Data Engineering*, 18(10), 2006.
3. C. Bornhövd, M. Altinel, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. DBCache: middle-tier database caching for highly scalable e-business architectures. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, 2003.
4. A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.
5. K. S. Candan, W. S. Li, Q. Luo, W. P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, 2001.
6. K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. *SIGMOD Record*, 30(2):532–543, 2001.
7. P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, 1997.
8. T. Corman, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT press Cambridge, MA, USA, 1990.
9. S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc. Int'l. Conf. on Very Large Databases*, 1996.
10. P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *Proc. 10th Int'l. World Wide Web Conf.*, 2001.
11. O. Deux et al. The story of O2. *Trans. on Knowledge and Data Engineering*, 2(1), 1990.
12. M. Garey and D. Johnson. *Computers and intractability: a guide to NP-completeness*. WH Freeman and Company, San Francisco, California, 1979.
13. C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable query result caching for web applications. In *Proc. Int'l. Conf. on Very Large Databases*, 2008.
14. H. Guo, P. Larson, R. Ramakrishnan, and J. Goldstein. Support for relaxed currency and consistency constraints in mtcache. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, 2004.
15. D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.
16. Y. Huang, R. Sloan, and O. Wolfson. Divergence Caching in Client Server Architectures. In *Proc. 3rd International Conference on Parallel and Distributed Information Systems*, 1994.
17. N. Kaiser. Pan-starrs: a wide-field optical survey telescope array. *Ground-based Telescopes*, 5489(1):11–22, 2004.
18. N. Kaiser et al. Pan-STARRS: a large synoptic survey telescope array. In *Proc. SPIE*, pages 154–164, 2002.
19. P. Kunszt, A. Szalay, and A. Thakar. The hierarchical triangular mesh. In *Mining the Sky: Proc. MPA/ESO/MPE Workshop*, 2001.

20. A. Labrinidis and N. Roussopoulos. Webview materialization. *SIGMOD Record*, 29(2), 2000.
21. A. Labrinidis and N. Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *The VLDB Journal*, 13(3), 2004.
22. C. Lecluse, P. Richard, and F. Velez. O2, an object-oriented data model. *SIGMOD Record*, 17(3):424–433, 1988.
23. Large Synoptic Survey Telescope. http://www.lsst.org.
24. T. Malik, R. Burns, and A. Chaudhary. Bypass caching: Making scientific databases good network citizens. In *Proc. Int'l. Conf. on Data Engineering*, 2005.
25. T. Malik, R. Burns, and N. Chawla. A black-box approach to query cardinality estimation. In *Proc. 3rd Conf. on Innovative Data Systems Research*, 2007.
26. T. Malik, X. Wang, P. Little, A. Chaudhary, and A. R. Thakar. Robust caching for rapidly-growing scientific repositories. http://www.cs.purdue.edu/ tmalik/Delta-Full.pdf, 2010.
27. C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. *ACM SIGMOD Record*, 30, 2001.
28. C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *CIDR*, 2005.
29. C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, 2002.
30. Pan-STARRS—Panoramic Survey Telescope and Rapid Response System. http: www.pan-starrs.ifa.hawaii.edu.
31. G. Peng. CDN: Content distribution network. *Arxiv preprint cs.NI/0411069*, 2004.
32. P. Protopapas, R. Jimenez, and C. Alcock. Fast identification of transits from light-curves. *Journal reference: Mon. Not. Roy. Astron. Soc*, 362:460–468, 2005.
33. Sloan Digital Sky Survey. www.sdss.org.
34. A. Shoshani, A. Sim, and J. Gu. *Storage Resource Managers: Essential Components for the Grid*. Kluwer Academic Publishers, 2004.
35. V. Singh, J. Gray, A. R. Thakar, A. S. Szalay, J. Raddick, B. Boroski, S. Lebedeva, and B. Yanny. SkyServer Traffic Report: The First Five Years, MSR-TR-2006-190. Technical report, Microsoft Technical Report, Redmond, WA, 2006.
36. S. Skiena. *The algorithm design manual*. Springer, 1998.
37. W. Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1993.
38. M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A new architecture for distributed data. In *Proc. of the Internationall Conference on Data Engineering*, 1994.
39. M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *The VLDB Journal*, 5(1), 1996.
40. A. S. Szalay, J. Gray, A. R. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg. The SDSS skyserver: public access to the Sloan Digital Sky Server data. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, 2002.
41. The Times Ten Team. In-memory data management in the application tier. In *Proc. of the International Conference on Data Engineering*, 2000.
42. X. Wang, T. Malik, R. Burns, S. Papadomanolakis, and A. Ailamaki. A workload-driven unit of cache replacement for mid-tier database caching. In *Advances in Databases: Concepts, Systems and Applications*, volume 4443, pages 374–385, 2007.
43. E. W. Weisstein. Vertex cover. from mathworld–a wolfram web resource. http://mathworld.wolfram.com/VertexCover.html.