A Workload-Driven Unit of Cache Replacement for Mid-Tier Database Caching

Xiaodan Wang[1], Tanu Malik[1], Randal Burns[1], Stratos Papadomanolakis[2], and Anastassia Ailamaki[2]

[1] Johns Hopkins University, USA
{xwang, tmalik, randal}@cs.jhu.edu
[2] Carnegie Mellon University, USA
{stratos, natassa}@cs.cmu.edu

Abstract. Making multi-terabyte scientific databases publicly accessible over the Internet is increasingly important in disciplines such as Biology and Astronomy. However, contention at a centralized, backend database is a major performance bottleneck, limiting the scalability of Internet-based, database applications. Mid-tier caching reduces contention at the backend database by distributing database operations to the cache. To improve the performance of mid-tier caches, we propose the caching of query prototypes, a workload-driven unit of cache replacement in which the cache object is chosen from various classes of queries in the workload. In existing mid-tier caching systems, the storage organization in the cache is statically defined. Our approach adapts cache storage to workload changes, requires no prior knowledge about the workload, and is transparent to the application. Experiments over a one-month, 1.4 million query Astronomy workload demonstrate up to 70% reduction in network traffic and reduce query response time by up to a factor of three when compared with alternative units of cache replacement.

1 Introduction

The sciences are collecting and analyzing vast amounts of observational data. In Astronomy, cataloging and mapping spectral characteristics of objects in only a fraction of the sky requires several terabytes of storage. Data are made available to remote users for processing, for example through SkyQuery [1], a federation of Astronomy databases and part of the World-Wide Telescope [2]. However, SkyQuery faces an impending scalability crisis. The federation is expected to expand from roughly a dozen members today to over a hundred in the near future [3]. Furthermore, member databases, such as the Sloan Digital Sky Survey (SDSS) [4], are accumulating data at an astonishing rate.

Mid-tier caching is an attractive solution for increasing scalability, availability, and performance of distributed database applications [5]. We study mid-tier caching in the context of SkyQuery using bypass-yield caching [6]. Bypass-yield caching replicates database objects, *e.g.* columns (attributes), tables, or views, at caches deployed near the clients so that queries are served locally, reducing network bandwidth requirements. Caches service some queries in cache and ship other queries to be evaluated at the backend database.

Our experience with bypass-yield caching indicates that query evaluation performance in the cache is also critical. Despite the network benefits, poor I/O performance in caches may result in inferior overall performance. Mid-tier caches lack the indices that are vital to I/O efficiency in databases. Maintaining indexes in a cache is prohibitively expensive given that (1) index construction is time consuming and I/O-intensive, (2) cache data are continuously changing, and (3) indices consume space, polluting the cache with replicated data. In this paper, we extend previous work on network traffic reduction with bypass-yield caching [6] by exploring ways to simultaneously improve query execution performance in the cache.

In existing mid-tier caching models, the storage organization employed by the cache is either tied to the backend database or defined *a priori*, *e.g.* columns [6], tables [5], vertical or horizontal fragments of base tables [7,8], or views [9]. Our work differs from previous caching approaches in two ways. First, we explore *dynamic* cache storage organizations that take into account workload information to improve query performance. Second, we evaluate alternative units of cache replacement in terms of their network traffic reduction benefits.

We propose a workload-driven technique for choosing the unit of cache replacement that is adaptive and self-organizing. Our model employs query prototypes in which each prototype is a combination of attributes that is accessed by the same class of queries. Prototypes serve as the logical unit of cache replacement. Query prototypes are adaptive in that prototypes are defined dynamically based on the access pattern of queries that appear in the workload. This is useful for scientific databases in which an *a priori* workload is not available. In particular, Astronomers are constantly finding new experiments to conduct in SkyQuery, making it difficult to identify a static set of frequently accessed database objects. Query prototypes are self-organizing in that changes to the storage organization are part of the cache replacement decision. Each prototype is optimized for a specific class of queries and, as workloads change, the storage layer changes accordingly. This makes it unnecessary to reorganize the cache contents periodically to improve query performance.

Our experiments show that query prototypes result in a factor of three reduction in query response time when compared with caching of columns, tables, and vertical partitions of backend tables. Prototypes also exhibit low cache pollution and high network savings. This is especially true at low cache sizes in which 40% less network traffic was generated when compared with the next best method.

We emphasize that this paper does not introduce a new caching algorithm, but presents a technique for specifying the unit of cache replacement that improves performance without sacrificing the inherent merits of mid-tier database caching.

2 Caching for Scientific Databases

We briefly describe the framework used to study our approach and explain why choosing the unit of cache replacement is relevant to database caching.

2.1 Cache Environment

SkyQuery [1], a federation of Astronomy databases, is a Web-based application in which caching drastically reduces network traffic. In SkyQuery, Web mediators or portals located in close proximity to the users serve as the interface between user queries

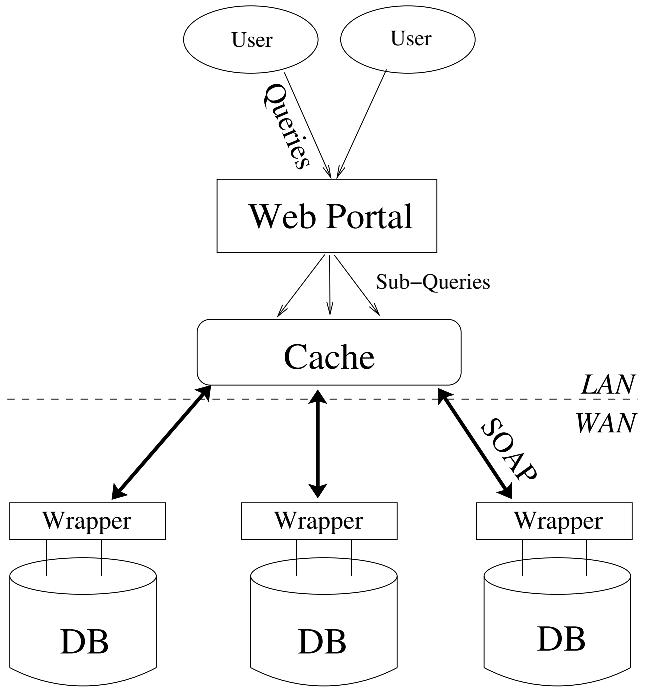

Fig. 1. Mid-tier database caching in SkyQuery

and member databases in the federation. As shown in Figure 1, the portal communicates with member databases via a wrapper interface. The wrapper interface allows member databases to remain autonomous and heterogeneous entities. We are currently building a cache prototype in SkyQuery in which the cache resides at the portal and utilizes the wrapper interface to transfer data, process queries, and collect schema information.

The SkyQuery workload is read-only and contains a rich variety of range, aggregate, identity, and spatial queries submitted through a large community of Astronomers. User queries either execute locally in the cache or are shipped to backend databases. However, executing queries at the backend database generates a lot of network traffic over WAN, because query results are transferred back to the user. The goal is to cache database objects so that most of the data transfer is from the cache to the user over LAN.

We employ bypass-yield caching (BYC) [6], which was developed for SkyQuery. The primary goal of caching in this environment is network traffic reduction because queries are network-bound [6]. BYC is an economic framework in which network bandwidth is the currency and network traffic reduction is the goal. The decision to ship a database object to the cache represents an investment in which the cost to load an object is recovered through expected network savings. Queries that access objects which fail to yield positive network savings in this economic model are bypassed, *i.e.* shipped to the backend database for execution. BYC is also flexible with respect to the unit of cache replacement; replacement can be performed on individual columns, tables, or tuples.

Query prototypes is an extension to BYC by making storage management part of the cache replacement decision. Storage management is important because mid-tier caches operate in index-free environments in which cache data are continuously changing and cache space is constrained. While BYC identifies data that are beneficial to the cache, it does not consider how data layout on disk impacts query execution performance. Query prototypes not only capture data that are useful to cache but also how data should be organized on disk.

2.2 Choosing the Unit of Cache Replacement

The granularity of cache replacement has a significant impact on the overall network performance [6]. Network performance is governed by the utilization of cache space. In general, caching objects at too fine a granularity increases maintenance overhead. For example, if data granularity is chosen at the level of individual tuples, then significant cache space is needed to maintain the relationships among all of the cached tuples. On the other hand, too coarse a granularity degrades cache utilization by emptying a large portion of the cache during object replacement [10].

Query performance is governed by the effective clustering of data in the cache. The best clustering is obtained if groups of workload-related tuples are cached, as in semantic caching. However, semantic caching requires that workloads exhibit query locality, which is not true of Astronomy workloads [6]. Furthermore, splitting, coalescing, and containment checking is difficult when workloads consist of complex queries, and not just range queries [7]. Thus, from the perspective of query performance, cache replacement must be performed at the granularity of tables, columns, or vertical or horizontal fragments of the backend database. However, these database objects are defined during the database design process, which is concerned with eliminating redundancy and update anomalies in addition to workload access patterns. Naively choosing any of the above as the choice of caching granularity forces columns that are logically related but rarely accessed together to be stored together in the cache. This hurts both query and network performance. In contrast, if the unit of cache replacement is adaptive so that columns are grouped in a manner that reflects changing query access patterns, then overall cache performance is improved.

3 Related Work

In this section, we summarize work on mid-tier caches that define static and dynamic units of cache replacement. We also review several database design methods for improving query performance, including vertical partitioning and materialized views.

3.1 Statically-Defined Cache Replacement

Mid-tier caching provides greater scalability and availability, increased performance, and quality of service guarantees [5]. Several high performance mid-tier caching systems allow for flexibility in defining the unit of cache replacement [7,8,9]. Cache Tables [7] allows for the caching of *declarative cache tables*, which corresponds to a table, column, or materialized view from the backend database. Similar flexibility in the unit of cache replacement is achieved in TimesTen [9] through the definition of *cache groups* and MTCache [8] through the use of *select-project views*. However, the unit of cache replacement in these systems is specified *a priori* during initialization. Also, the unit of cache replacement is static and does not adapt to workload changes. We found that adapting the storage organizations of cached contents by changing the unit of cache replacement over time significantly improves query performance.

3.2 Dynamically-Defined Cache Replacement

Predicate-based or semantic caching supports an adaptive unit of cache replacement [11]. Caching is performed on groups of spatially related tuples, as defined by a set of predicate conditions, which exhibit high semantic reuse. Unlike statically-defined caching schemes, which are susceptible to poor clustering, semantic regions can grow or shrink in size to adapt to workload changes. Judicious data placement, accomplished by preserving spatial locality of data that are frequently accessed together, improves query performance significantly over static schemes [10].

While semantic caching is attractive, there are some drawbacks. Maintaining a semantic description of the data is feasible when workloads consist mostly of simple *select-project-join* queries [7]. Also, workloads should exhibit semantic reuse in which data items contained in the result of a query are later reused. These properties allow for efficient checking of query containment against cached data [11]. However, Astronomy workloads comprise of nested queries with user-defined functions and complex joins, which are unsuitable for semantic caching. More importantly, scientific workloads exhibit little semantic reuse but frequent syntactic reuse [6]; *i.e.* specific data items experience little to no reuse, but queries request data from the same group of columns. We exploit syntactic reuse by caching dynamically-defined groups of columns.

3.3 Database Design Methods

Vertical partitioning identifies I/O efficient placement of columns at the storage layer to improve query performance [12,13,14,15,16,17]. Early works introduced the notion of affinity, the frequency in which attributes are accessed together, to evaluate the placement of columns [12,13,18]. Data columns are grouped together by applying a clustering algorithm on affinity values. Recent work suggests that affinity value is decoupled from actual I/O cost and is a poor predictor of query performance. They propose more sophisticated cost models that estimate the I/O cost of performing database operations [14,15]. AutoPart [15] is a workload-based, automated database design tool that improves query performance through vertical and horizontal partitioning. While query prototypes can be described as a vertical fragmentation of the database, it does not partition the database algorithmically. Also, prototypes are not disjoint in that multiple prototypes may replicate the same data column.

Materialized views allow for arbitrary vertical and/or horizontal fragmentations of base tables. Materialized views are concrete tables derived from underlying base relations to enable I/O efficient accesses. In ViewCache, a framework for managing materialized views is provided that balances storage overhead with performance [19]. Views are made compact by storing pointers to records in the base relation and are materialized on demand. However, ViewCache does not specify the appropriate views to create.

Multi-query optimization, a technique used in data warehouse systems, provides one solution to view selection [20]. The goal is to exploit shared data between a set of queries or views and identify additional views for transient or permanent materialization that are used to share intermediate results and improve performance. Multi-query optimization has been successfully applied in view maintenance [21] and in the optimization of inter-query execution [22]. However, evaluating shared sub-expressions increases the complexity of query optimization. Query prototype caching does not increase optimization cost because each query is evaluated against a single prototype. Moreover, multi-query optimization is applied to known workloads that are fairly static and that have high overlap across queries. Neither are assumptions for query prototypes.

4 Query Prototype Caching

This section provides a formal description of query prototypes. The method for specifying query prototypes takes as input a set of queries ***Q*** and outputs a set of prototypes ***P***, which serve as the unit of cache replacement. Each query is matched against exactly one prototype, whereas each prototype is derived from a set of related queries.

4.1 Definition

Let ***R*** = R_1, ..., R_n be the set of all tables at the backend database. Each table consists of a set of attributes. Let ***A*** = A_{11}, ..., A_{1m}, A_{21}, ..., A_{np} as the set of all attributes at the backend database in which A_{ij} is the j^{th} attribute in relation R_i.

Let ***Q*** be the set all queries in the workload in which $q_i \in \boldsymbol{Q}$ is the i^{th} query in the workload. AutoPart [15] introduced the concept of a *Query Access Set* (QAS), which is the subset of attributes from a *single* relation in ***R*** that are referenced by a query in ***Q***. For query prototypes, we redefine *Query Access Set* to be the set of attributes from *every* relations in ***R*** that are referenced by a query in ***Q***.

Let QAS$(q_i, \boldsymbol{A})$ be defined as the set of attributes from ***A*** that are referenced by query q_i. For query prototype caching, we consider queries q_i and q_j to be equivalent if and only if QAS$(q_i, \boldsymbol{A})$ = QAS$(q_j, \boldsymbol{A})$ – that is, they access the same set of attributes. A set of queries, in which the QAS of these queries are identical, make up an equivalence class in the workload. Each prototype in ***P*** represents a unique equivalence class in the workload. Thus, to cache a prototype P_k, the set of attributes referenced by queries in P_k are loaded into the cache as one unit.

We demonstrate the concept of query prototypes using three queries derived from an Astronomy workload [4]:

q_1: SELECT objID FROM Galaxy, SpecObj
WHERE objID = bestobjID and specclass = 2 and z between 0.121 and 0.127
q_2: SELECT objID, ra, dec FROM PhotoPrimary WHERE dec between 2.25 and 2.75
q_3: SELECT top 1 ra, dec FROM PhotoPrimary WHERE objID = 5877311875315

The example shows two unique query prototypes in which the *Query Access Set* for q_2 and q_3 are identical:

QAS$(q_1, \boldsymbol{A})$ = {Galaxy:objID, SpecObj:bestobjID, SpecObj:specclass, SpecObj:z}
QAS$(q_2, \boldsymbol{A})$=QAS$(q_3, \boldsymbol{A})$={PhotoPrimary:objID, PhotoPrimary:ra, PhotoPrimary:dec}

4.2 Discussion

To materialize a prototype in the cache, the group of attributes belonging to the prototype is allocated a unique set of relation files in the storage layer. Spatial locality among attributes belonging to each prototype is preserved because storage is not shared between prototypes with overlapping attributes. Queries serviced from the cache are matched against a prototype and are rewritten to address relations in the cache. Furthermore, loading or evicting a prototype simply requires that the corresponding relations be added or dropped from the database.

Two properties are worth noting for query prototypes. Prototypes are rarely disjoint; attributes appear in multiple prototypes. This introduces attribute replication because prototypes that overlap do not share storage. A prototype can also contain attributes from multiple tables. One option for storing attributes from multiple tables computes the cross product and stores the result in a single table. However, this utilizes cache storage unwisely, because the storage required scales exponentially with the number of joins [23]. Instead, we store attributes belonging to different backend relations in separate tables and compute the join during query execution.

Theoretically, the number of query prototypes can be very large, equal to $min($# of queries, 2^n where n is the number of attributes referenced by the workload$)$. In practice, query prototypes provide a compact summary of the workload, even those with millions of queries. We found that much of the science is conducted through prepared SQL queries via form-based applications or custom scripts that iterate over the database by accessing the same combination of attributes. This observation was exploited in a work on active, form-based proxy caching for backend databases [24]. Luo and Naughton acknowledged that while caching for queries with arbitrary SQL syntax is difficult, queries submitted through HTML forms exhibit only a handful of distinct types. The semantic information from these queries is captured through a compact set of *templates*, which are parameterized query specification in which parameter values are provided at runtime. Our approach applies to general queries and does not distinguish between form-based and arbitrary user queries.

4.3 Performance Implications

Query prototypes improve query performance at the cache by reducing the amount of data read from disk. Since prototypes contain only the attributes referenced by a query, accessing them is much faster than accessing the entire base table. Query prototypes reduce scan costs by ensuring that only data from columns requested by a query are read from disk. Also, each query is executed against a single prototype in which the number of join operations is never more than the number of backend tables a query accesses.

Besides improving performance for the queries executed in the cache, query prototypes perform well in terms of network bandwidth. Contrary to using entire tables as the cache replacement unit, query prototypes avoid unnecessary transfers of attributes that are not referenced by the workload. Although single-column partitions also have this property [6], they suffer from the *file-bundling* effect [25] in which evicting a single column from a group of columns that are accessed together renders the cache ineffective for the entire group.

Cache pollution is the loading of redundant or unused database objects that adversely impact performance by evicting useful objects. Pollution limits the applicability of query prototypes to general workloads. Since prototypes are not disjoint, the cache allocates storage for duplicate columns. If the number of prototypes is too high, then no single prototype serves enough queries to provide positive network savings. If the overlap between prototypes is high, then attribute replication quickly pollutes the cache.

5 Experiments

Our experiments use a one-month query trace from the Sloan Digital Sky Survey (SDSS), a major site in SkyQuery [4]. The trace consists of 1.4 million read-only, SQL queries generating nearly 360GB of network traffic against Data Release 4 (DR4), which is a two-terabyte database. To finish I/O experiments in a reasonable time, we take a ten percent sample of the DR4 database, roughly 200GB in size.

We evaluate query prototypes against three units of cache replacement: columns, tables, and logical groupings of columns as determined through vertical partitioning. For column caching, we store related columns in the same table rather than storing each column in a separate table. Query performance in the latter approach is disastrous using row-oriented, relational databases.

We adapt existing vertical partitioning algorithms to caching. Traditionally, partitioning takes as input a representative set of queries and outputs an alternative, I/O efficient database schema. Thus, partitioning algorithms are designed for a different set of goals than caching – that is, improving query performance rather than network usage. Nonetheless, we expect that the same technique for improving spatial locality among columns that are accessed together will group columns that are relevant to the cache. We choose AutoPart [15], a high-performance vertical partitioning algorithm, for our experiments. To ensure that the unit of cache replacement is adaptive, we periodically update the column groupings by rerunning the algorithm with new queries. We also restrict input to the algorithm to queries with results sizes greater than one megabyte because it is not economical to cache for queries with small result sizes.

All experiments use the DR4 database running on Microsoft's SQL Server 2000. Our main workstation is a 3GHz Pentium IV machine with 1GB of main memory and two SATA disks. SQL Server uses one disk for logging and stores the database on a second, 500GB disk.

5.1 Query Workload

Upon analyzing the DR4 trace, we discover that query prototypes provide a compact representation of the workload. The entire 1.4 million trace consists of only 1176 prototypes, owing to schema reuse in which a limited combination of columns are repeatedly accessed. Moreover, a handful of prototypes capture most of the workload. 11 unique prototypes describe 91% of the entire workload while 6 unique prototypes generate 89% of the network traffic. Query prototypes that occur frequently do not correspond to those that generate most of the network traffic. The latter dictate cache replacement decisions whereas the former do not.

5.2 Query Performance

We measure query performance by deploying the cache for the sampled database and using the entire 1.4 million query workload as input to the cache. The cache size is set at 1% of the sampled database. Using a small relative cache size is appropriate, because caches are likely to have only a fraction of the several terabytes of storage available to backend databases. Figure 2 shows the performance of queries executed at the cache. Caching query prototypes results in the best performance with an average query response time of 474ms, which is up to three times faster than other strategies.

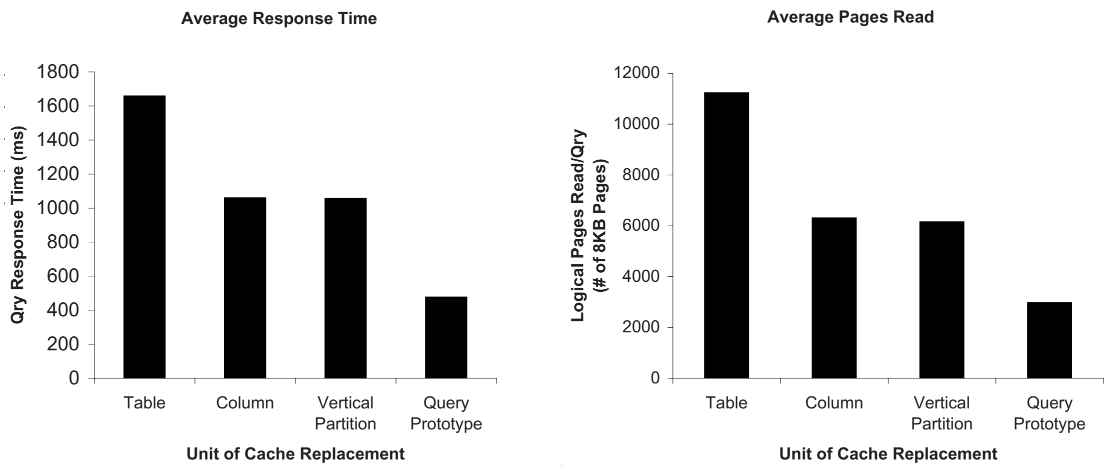

Fig. 2. Query performance by unit of cache replacement

The lack of prior knowledge about the workload limits the effectiveness of vertical partitioning. The partitioning algorithm takes all high-yield queries from the workload as input, but only a fraction of those queries are executed in the cache. This results in a mismatch between the workload executed in the cache and the workload provided as input to the partitioning algorithm.

We also measure logical database pages read to compare sequential access performance. Unlike physical reads, buffering and fragmentation on disk do not affect logical reads. The trend is consistent with query response times; query prototypes employ the most I/O-efficient layout of data and incur the fewest page reads.

5.3 Network Savings

In Figure 3, we compare network performance on the un-sampled database for different caching strategies at various increments of cache size. Query prototypes outperform column caching at cache sizes lower than 2% of the database, while table caching lags far behind. The enormous size of each table means that for cache sizes less than 0.5%, only a handful of objects fits in cache when table caching is used. As cache size increase, column caching exhibits a slight advantage over query prototypes. With the cache

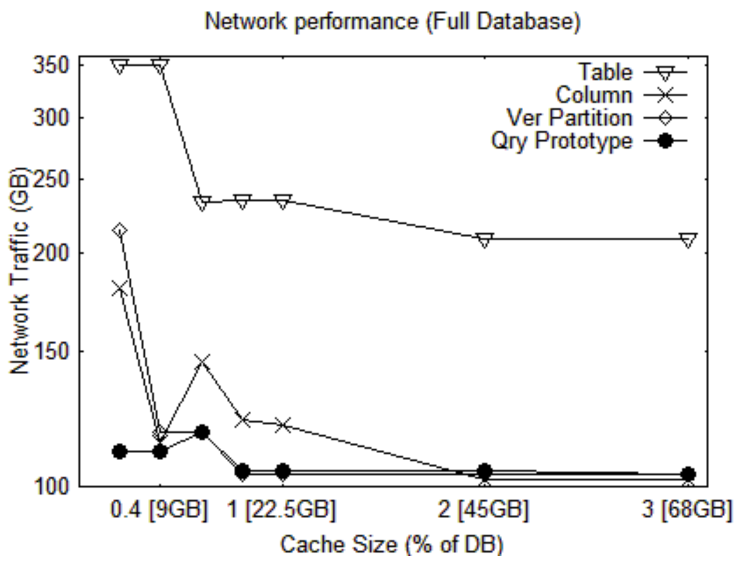

Fig. 3. Network traffic by unit of cache replacement. Network cost without caching is 357GB

at 3% of the database, query prototype caching generates 104GB of network traffic versus 102GB for column caching. Vertical partition caching also performs well and do a good job of grouping columns that provide positive network savings.

A conspicuous feature at cache sizes lower than 2% is that query prototype outperforms column caching by a large margin, generating up to 40% less traffic. This is explained by the file-bundling effect [25]. Specifically, column caching makes poor replacement decisions at low cache sizes when cache resident times are shorter and object evictions are more frequent. The resulting mix of columns in the cache have a lower probability of satisfying incoming queries.

5.4 Cache Pollution

In query prototypes, pollution arises from attribute replication. For table and vertical partition caching, columns that do not provide any network savings pollute the cache because they are grouped with columns that yield positive network savings.

Figure 4 shows cache pollution on the un-sampled database for different caching strategies. Attribute replication for query prototypes remains at 5% or lower for the most part. There is a sharp rise at 0.6% cache size, which does not significantly impact network savings because cache space is not fully utilized. Immediately after is a sharp drop because prototypes that were previously too large to fit into the cache are loaded. This displaces several smaller prototypes, resulting in fewer, large prototypes. Pollution remains fairly stable afterwards as the number of cached prototypes increases. Caching vertical partitions exhibit significantly more pollution than query prototypes. This is because partitioning algorithms are not designed for caching and do not always separate attributes that provide network savings from those that do not.

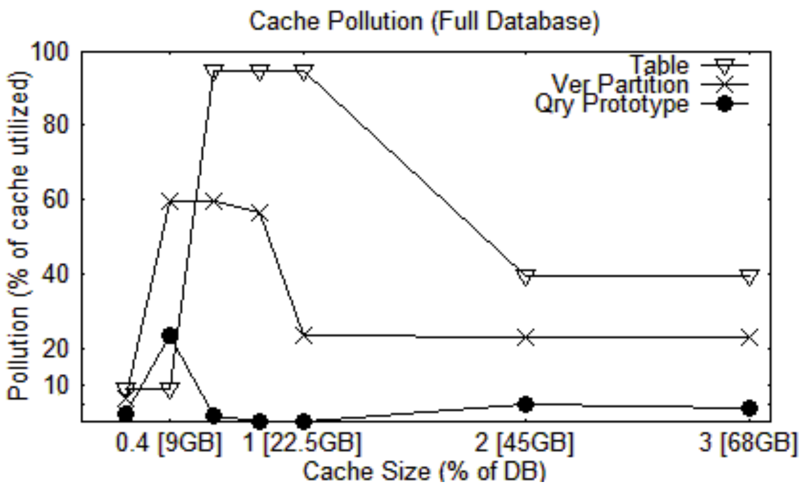

Fig. 4. Cache pollution, as a percent of cache space utilized, by unit of cache replacement

6 Conclusions and Future Work

We present a workload-driven approach to specifying the unit of cache replacement for scientific workloads that is adaptive and self-organizing. Our experiments illustrate that query prototypes achieve superior query and network performance with little cache pollution. However, prototypes are susceptible to cache pollution for workloads with high attribute overlap. To address this, we are considering merging prototypes with shared attributes in order to strike a balance between query performance and cache pollution. Vertical partition caching is promising because unlike query prototypes, it is not susceptible to a high degree of attribute replication. However, directly adapting existing partitioning algorithms to caching is unsuitable because these algorithms optimize query execution cost but do not consider network costs. We will extend partitioning algorithms to optimize for multiple costs in the future. Finally we plan to extend query prototypes to support updates.

References

1. Malik, T., Szalay, A.S., Budavri, A.S., Thakar, A.R.: SkyQuery: A Web Service Approach to Federate Databases. In: CIDR. (2003)
2. Gray, J., Szalay, A.: Online Science: The World-Wide Telescope as a Prototype for the New Computational Science. Presentation at the *Supercomputing Conference* (2003)
3. Szalay, A., Gray, J., Thakar, A., Kuntz, P., Malik, T., Raddick, J., Stoughton, C., Vandenberg, J.: The SDSS SkyServer - Public Access to the Sloan Digital Sky Server Data. In: SIGMOD. (2002)
4. The Sloan Digital Sky Survey. http://www.sdss.org
5. Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B.G., Naughton, J.F.: Middle-tier Database Caching for E-Business. In: SIGMOD. (2002)
6. Malik, T., Burns, R., Chaudhary, A.: Bypass Caching: Making Scientific Databases Good Network Citizens. In: ICDE. (2005)

7. Altinel, M., Bornhvd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., Reinwald, B.: Cache Tables: Paving the Way for An Adaptive Database Cache. In: VLDB. (2003)
8. Larson, P., Goldstein, J., Guo, H., Zhou, J.: MTCache: Mid-Tier Database Caching for SQL Server. In: ICDE. (2004)
9. The TimesTen Team: Mid-tier Caching: The TimesTen Approach. In: SIGMOD. (2002)
10. Dar, S., Franklin, M.J., Jonsson, B.T., Srivastava, D., Tan, M.: Semantic Data Caching and Replacement. In: VLDB. (1996)
11. Keller, A.M., Basu, J.: A Predicate-based Caching Scheme for Client-Server Database Architectures. VLDB (1996)
12. Hammer, M., Niamir, B.: A Heuristic Approach to Attribute Partitioning. In: SIGMOD. (1979)
13. Navathe, S., Ceri, S., Wiederhold, G., Dou, J.: Vertical Partitioning Algorithms for Database Design. ACM Trans. Database Syst. **9**(4) (1984) 680–710
14. Chu, W.W., Ieong, I.T.: A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems. IEEE Trans. Softw. Eng. **19**(8) (1993) 804–812
15. Papadomanolakis, S., Ailamaki, A.: AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In: SSDBM. (2004)
16. Cornell, D.W., Yu, P.S.: An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. IEEE Trans. Softw. Eng. **16**(2) (1990) 248–258
17. Agrawal, S., Narasayya, V.R., Yang, B.: Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In: SIGMOD. (2004)
18. Navathe, S.B., Ra, M.: Vertical Partitioning for Database Design: A Graphical Algorithm. In: SIGMOD. (1989)
19. Roussopoulos, N.: An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. ACM Trans. Database Syst. **16**(3) (1991) 535–563
20. Sellis, T.K.: Multiple-Query Optimization. ACM Trans. Database Syst. **13**(1) (1988) 23–52
21. Mistry, H., Roy, P., Sudarshan, S., Ramamritham, K.: Materialized View Selection and Maintenance Using Multi-Query Optimization. In: SIGMOD. (2001)
22. Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and Extensible Algorithms for Multi Query Optimization. In: SIGMOD. (2000)
23. Ioannidis, Y.E., Christodoulakis, S.: On the Propagation of Errors in the Size of Join Results. In: SIGMOD. (1991)
24. Luo, Q., Naughton, J.F.: Form-Based Proxy Caching for Database-Backed Web Sites. In: VLDB. (2001)
25. Otoo, E., Rotem, D., Romosan, A.: Optimal File-Bundle Caching Algorithms for Data-Grids. In: ACM/IEEE Supercomputing (SC). (2004)