Automated Physical Design in Database Caches

Tanu Malik[1], Xiaodan Wang[1], Randal Burns[1], Debabrata Dash[2], Anastasia Ailamaki[2]

[1] *Johns Hopkins University, USA*
{tmalik,xwang,randal}@cs.jhu.edu

[2] *Carnegie Mellon University, USA*
{ddash,natassa}@cs.cmu.edu

***Abstract*— Performance of proxy caches for database federations that serve a large number of users is crucially dependent on its physical design. Current techniques, automated or otherwise, for physical design depend on the identification of a representative workload. In proxy caches, however, such techniques are inadequate since workload characteristics change rapidly. This is remarkably shown at the proxy cache of SkyQuery, an Astronomy federation, which receives a continuously evolving workload. We present novel techniques for automated physical design that adapt with the workload and balance the performance benefits of physical design decisions with the cost of implementing these decisions. These include both competitive and incremental algorithms that optimize the combined cost of query evaluation and making physical design changes. Our techniques are general in that they do not make assumptions about the underlying schema nor the incoming workload. Preliminary experiments on the TPC-D benchmark demonstrate significant improvement in response time when the physical design continually adapts to the workload using our online algorithm compared with offline techniques.**

I. Introduction

The performance of a database application crucially depends on the underlying physical design of the database. Physical design is often determined using a representative workload. However, many applications such as proxy caches [1][2][3] and content distribution networks [4] serve a continuous stream of queries from tens and thousands of users such that identifying a representative workload is difficult. Workload-based physical design techniques [5][6][7] are useful additions to these applications because they enable the exploration of physical design alternatives that suit the current workload. However, current techniques, automated or otherwise, are offline in nature: they are invoked by the system administrator using a representative workload and provide a static design for the entire workload. Online physical design techniques benefit database applications by detecting changes in the workload and adapting the physical design automatically.

We are particularly interested in the physical design of Bypass cache [8], a proxy database cache for the SkyQuery [9] federation of Astronomy databases. In fact, performance of Bypass cache critically depends upon its physical design. Bypass caches store database objects such as tables or columns close to users, providing dramatic reduction in both network traffic and query response time [8]. However, the physical design of cached objects is static in that it mirrors the design at the backend databases. For instance, columns belonging to the same physical table in the backend database are also stored together in the cache. Grouping columns in this manner translates into poor query execution performance in the cache, which may offset the response time benefit of serving queries locally.

Several techniques such as vertical partitioning [10][11] and index selection [5] can be applied to improve the physical design. Vertical partitioning, in particular, is an attractive solution for improving the physical design of Bypass caches. Construction of auxiliary data structures such as indices and materialized views presents a trade-off between the allocation of cache space for creating auxiliary structures or for caching more data. Vertical partitioning achieves performance benefits for queries by grouping columns from the same logical table into separate, non-overlapping physical relations. Thus, vertical partitioning reduces the amount of I/O incurred without introducing any redundant data that compete for cache space. However, the application of current vertical partitioning techniques to SkyQuery is difficult because they require the identification of a representative workload. Finding a representative workload in SkyQuery is hard; Astronomy workloads exhibit considerable *evolution* (*i.e.* the manner in which attributes are grouped by queries can change drastically within a week).

In this paper, we analyze physical design issues associated with Bypass caches and develop algorithms for online vertical partitioning that are sensitive to workload evolution. Using Bypass caches as a case study, we examine its sensitivity to physical design decisions, determine the degree of evolution in the workload, and quantify the cost of offline vertical partitioning in a cache. We also describe an online vertical partitioning solution that is inspired by task systems [12]. Task systems are general systems that capture the cost of transitioning between two states in addition to the cost of executing a task in a given state. The transition cost prevents oscillations into states that are sub-optimal in the long run.

Our contributions include a three-competitive algorithm when there are only two alternatives for grouping attributes. We also provide a workload-adaptive, incremental algorithm for vertical partitioning when there are N possible alternatives. Both algorithms minimize the combined cost of query execution and of making physical design changes. Our algorithms are general in that they can improve the physical design, using vertical partitioning, of proxy database caches without making assumptions about the underlying physical design or the incoming workload. As a first step towards evaluating these

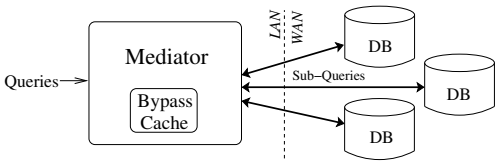

Fig. 1. Proxy caching in SkyQuery.

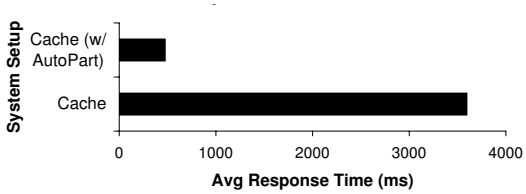

Fig. 2. Average response time of queries.

algorithms, we implement them using queries from the TPC-D benchmark. We compare our solution with a workload-based, offline vertical partitioning tool. Our experiments show a 17% improvement in average query response time when a single relation is continually reorganized in an online manner that adapts to workload changes. We are currently implementing our solution within SkyQuery and expect similar performance benefits.

II. Vertical Partitioning for Bypass Caching

In this section we first describe the Bypass cache application [8] and explain why caching provides an interesting case study for physical design. We then consider the advantages of physical design and examine the degree of evolution in the workload.

Bypass caches are proxy caches that reside at the mediator site in the SkyQuery federation (Figure 1). A database cache is manifested for each participant in the federation. Queries are submitted to the mediator which divides them into sub-queries for member databases. Each sub-query is either satisfied locally at the cache or bypassed to the remote database. A query is satisfied if all columns [1] that it accesses are cached, else it is bypassed. Our vertical partitioning module is collocated at the cache and suggests physical design changes independently for each sub-database. The module sees only a subset of queries that are received by the cache, since some are bypassed due to caching decisions.

Databases in the SkyQuery federation possess complex schema designs that often comprise of several star-schemas. The database schema is fixed after the initial public release of the data [13] and remains static thereafter. Thus, any changes to the schema have to be done outside the repository. Most relations in the published schema are bulky in that hundreds of columns are grouped together. For instance, two frequently accessed relations, $PhotoObjAll$ and $Field$, consist of 446 and 422 columns respectively. While columns belonging to the same relation are logically related, Astronomy queries do not use all columns together. Moreover, the subsets of columns that are used together by queries change over time such that making workload-adaptive physical design decisions become difficult. Figure 9 in the Appendix shows the most frequent queries from the SkyQuery workload for three consecutive weeks. The columns that are accessed together during each week change and differ drastically from the grouping present in the original schema.

[1]Bypass caches have lower response times when columns, instead of tables, are cached.

We quantify the impact of vertical partitioning in Bypass caches by measuring improvements in query response time. Figure 2 compares the performance of the cache for a given workload in two instances: when vertical partitioning is not applied and the cached columns are grouped according to the physical design at the backend database, and when the columns are grouped using AutoPart [10], an offline, workload-based tool for vertical partitioning. For this experiment we maintain the cache at 30% of the database size and use a month long workload (February 2006) from the Sloan Digital Sky Survey (SDSS) [14], a participant in the SkyQuery federation. In Figure 2, AutoPart illustrates the advantage of vertical partitioning when performed in an online manner. To suit this experiment for Bypass caches, we use AutoPart in an online manner by re-partitioning the cached objects prior to each incoming query using the most recent, *single* query as input to AutoPart. The result demonstrates a significant improvement in response time for Bypass caches that is solely attributed to online vertical partitioning. It does not take into account the cost of implementing physical design changes nor the overhead of running an offline partitioning tool. These costs become significant if the column groupings change drastically over time. While column groupings are re-evaluated on a per query basis in Figure 2, our next experiment examines the periodicity and frequency at which column groupings change.

We take the SDSS workload and plot an affinity matrix of the column groupings. The basic premise is that columns that occur together and have similar frequencies should be grouped together in the same relation [15]. In Figure 3, we show the affinity matrix for ten attributes from a single table in which each grid entry corresponds to the frequency with which a pair of attributes are accessed together (ordering of attributes are the same along the row and column). Figure 3 demonstrates that column groupings change on a weekly basis. This means that re-partitioning columns over time can benefit query performance in Bypass caches. While a static analysis of the workload shows that groupings change on a weekly basis, an entirely different workload may reflect changes within the span of a day. Thus, it is difficult to ascertain a fixed time span for regrouping columns. An online algorithm, which weighs the benefits amongst various column groupings with each incoming query, can decide *when* to make this change. Section IV provides a formal framework for making online decisions.

An online vertical partitioning problem should include the *transition cost*; that is, the cost of changing the physical

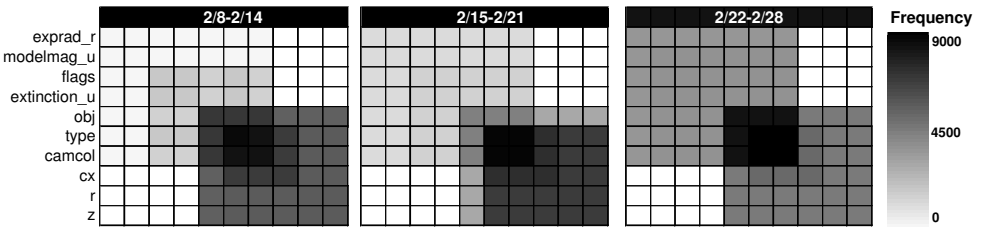

Fig. 3. Affinity matrix (co-access frequency) for ten select attributes from the $PhotoPrimary$ table.

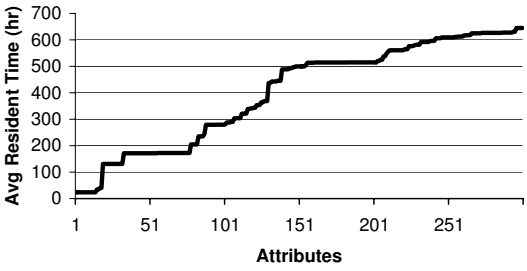

Fig. 4. Average cache resident times.

organization of columns through vertical partitioning [11][16]. Similar to query execution cost, transition cost can be quantified in terms of I/O. In Bypass caches, this cost should be considered along with the cache resident time of each column, which varies significantly across columns (Figure 4). For instance, there is little advantage to reorganizing short-lived columns, while other columns can amortize the transition cost over a longer period of time. The result from Figure 4 shows that many columns fall in the latter category.

III. Related Work

Physical design is defined a priori in several proxy caching systems [1][2][17]. In Cache Tables [2] a table, column, or materialized view is declared using the *declarative cache tables* construct. Similar flexibility is available in TimesTen [17] through the definition of *cache groups* and MTCache [1] through the use of *select-project views*. The schema elements that are materialized in the cache are specified during cache initialization, and do not adapt to workload changes.

Vertical partitioning groups columns that are accessed together in order to improve memory and disk performance [10][11][15][18][19][20]. Early work [15] derived affinity measurements from the workload as an indicator for grouping columns together. Columns are grouped by applying clustering algorithms on the affinity values. However, affinity values are decoupled from actual I/O cost, and thus are poor predictors of query performance. Recently, cost estimates from the optimizer or analytical cost-based models that capture the I/O of database operations are used to evaluate vertical partitions [10][11][19]. For instance AutoPart[10] interfaces with a commercial optimizer to obtain cost estimates for queries. However, existing solutions are offline, requires a representative workload, and provides a single, static physical design for the entire workload.

Identifying a representative workload is easy in applications in which the workload is fairly stable or is template-based [3] (*i.e.* users generate queries from pre-defined templates). However, these properties do not exist in SkyQuery for Astronomy workloads. Even if a representative workload is found, the process of evaluating when to run physical design tools is DBA dependent. Current research emphasizes the need for design tools that are *always-on* and can continuously adapt the physical design to changes in the workload [7]. Such tools have been studied for index selection [16], but not for vertical partitioning.

IV. Online Vertical Partitioning

We provide a formulation for online vertical partitioning that captures the cost of implementing physical design decisions in addition to query evaluation. Let $\mathbf{Q} = \{q_1, \ldots, q_m\}$ be an online sequence of queries, and let $\mathbf{C} = \{c_1, \ldots, c_n\}$ be the set of possible vertical partitions. In this section, we refer to each vertical partition $c_x \in \mathbf{C}$ as a *configuration*. Let a query q_i incur cost $q_i(c_x)$ if evaluated in configuration $c_x \in \mathbf{C}$, and let a transition from configuration $c_x \in \mathbf{C}$ to $c_y \in \mathbf{C}$ incur cost $d(c_x, c_y)$. Finally, let ϕ be a function $[1, m] \rightarrow \mathbf{C}$ in which $\phi(i)$ is the database configuration prior to the evaluation of q_i.

Given a database, a cache space constraint, and an initial configuration s, the goal of the online vertical partitioning problem is to find a ϕ that minimizes the cost of processing $\mathbf{Q}$:

$$cost(\phi, \mathbf{Q}) = \sum_{i=1}^{n} q_i(\phi(i)) + d(\phi(i-1), \phi(i))$$

in which $\phi(0) = s$.

The above formulation is similar to that of *task systems* introduced by Borodin et al. [12]. Task systems have been researched extensively, particularly when the transition costs form a metric [21][22]. Our costs are not symmetric and do not form a metric; that is, provided configurations c_x and c_y, $d(c_x, c_y)$ is not necessarily equivalent to $d(c_y, c_x)$. This is true because the sequence of operations (*i.e.* additions or deletions of tables or columns) required for making physical design changes in a database exhibits different costs.

An online algorithm ALG chooses from configurations in $\mathbf{C} = \{c_1, \ldots, c_n\}$ without seeing the complete workload $\mathbf{Q} = \{q_1, \ldots, q_m\}$. In particular, prior to evaluating query q_k, it decides on a configuration from $\mathbf{C}$ using only knowledge of queries $\{q_1, \ldots, q_k\}$. ALG is said to be α-competitive if there exists a constant b such that for every finite query sequence $\mathbf{Q}$,

```
2Conf(q_k :query)
// initially δ_max(0) = 0, C =current config,
// C̄ = opposite config
01  π(k) = q_k(C) - q_k(C̄)
02  δ_max(k) = max{π(k), δ_max(k-1) + π(k)}
03  if δ_max(k) > D
04    δ_max(k) = 0
05    Transition to C̄
```

Fig. 5. Online Algorithm for Two Configurations.

$cost(\phi_{ALG},\mathbf{Q}) \leq \alpha * cost(\phi_{OPT},\mathbf{Q}) + b$. OPT is the offline optimal that has complete knowledge of $\mathbf{Q}$.

In the remainder of this section, we describe a three-competitive online algorithm for the two-configuration scenario and extend our solution to N-configurations. For simplicity, our analysis is restricted to the partitioning of a single table T.

A. Two-Configuration Scenario

Given a relation T with n attributes, we restrict the solution to two configurations: configuration S in which each attribute is stored in a *separate* physical relation, and configuration M in which all attributes are *merged* into a single physical relation. In terms of query evaluation, S reduces the cost of scanning unused attributes while M minimizes join overhead for queries accessing many attributes. We make no assumptions about the transition costs $d(S, M)$ and $d(M, S)$.

Borodin et al.[12] give a general algorithm for metrical task systems that can be extended to task systems that are non-metric. We present a three-competitive algorithm $2Conf$ designed specifically for the two-configuration scenario. Our main observation is that $2Conf$ should transition to the opposite configuration if remaining in the current configuration incurs a substantial amount of "extra cost" from query execution.

Let D denote $d(S, M) + d(M, S)$, q_k be the current query, and C and $\overline{C}$ denote the current and opposite configurations respectively. The *penalty* $\pi(k)$ of remaining in C is defined as $q_k(C) - q_k(\overline{C})$ (*i.e.* the difference between the cost of evaluating q_k in C compared with the opposite configuration $\overline{C}$). Let q_i be the earliest query prior to q_k such that no transition occurs from q_i onwards. For any j in which $i \leq j \leq k$, define *cumulative penalty* $\delta(j, k)$ as $\sum_{x=j}^{k} \pi(x)$. Further, define the maximum cumulative penalty $\delta_{\max}(k)$ at q_k as $\max_{i \leq j \leq k} \delta(j, k)$. We are interested in the cumulative penalty incurred by a contiguous sequence of queries from q_j to q_k such that $\delta(j, k)$ is maximized. In other words, for all j' in which $j \leq j' \leq k$, $\delta(j, j') \geq 0$.

The pseudo-code for $2Conf$ is provided in Figure 5. $2Conf$ makes a transition before evaluating the current query q_k if $\delta_{\max}(k) > D$.

Theorem 1: $2Conf$ is three-competitive.

Proof: Consider any finite query sequence $\mathbf{Q}$. in which the ith query is q_i. Modify $\mathbf{Q}$ into the sequence $\mathbf{Q'}$: the ith query q'_i corresponds to q_i and has costs $q'_i(S) = \max\{q_i(S) - q_i(M), 0\}$ and $q'_i(M) = \max\{q_i(M) - q_i(S), 0\}$. In other words, q'_i subtracts the cost of q_i in each configuration by the cost of q_i in the cheapest configuration. This means that under $\mathbf{Q'}$, cumulative penalty is monotonically increasing and more transitions are incurred. It is easy to verify that the ratio $cost(\phi_{ALG}, \mathbf{Q'})/cost(\phi_{OPT}, \mathbf{Q'})$ is at least as large as the corresponding ratio with respect to $\mathbf{Q}$. For the rest of the proof we consider $\mathbf{Q'}$ instead of $\mathbf{Q}$.

In $\mathbf{Q'}$, there can be queries in which $2Conf$ happens to be in the cheaper configuration. For such queries, $2Conf$ incurs no cost and the OPT may incur a cost depending on the configuration it is in. In our accounting of the costs, we assume that OPT incurs no cost for such queries, irrespective of the configuration it is in.

Without loss of generality, assume $2Conf$ starts in S. Let it make $k \geq 0$ (k is even) transitions before ending at configuration S. After transition k, $2Conf$ incurs $k/2$ cycles of migrating from configuration S to M and back, resulting in a total transition cost of $(k/2)(d(S, M) + d(M, S))$. Additionally, $2Conf$ incurs a total query evaluation cost of $k(d(S, M) + d(M, S))$, which follows from the definition of $2Conf$ in which a transition occurs if maximum cumulative penalty exceeds $d(S, M) + d(M, S)$. Thus, $2Conf$ incurs cost $(3k/2)(d(S, M) + d(M, S))$ after k transitions.

To lower bound the cost incurred by OPT during the the first k transitions of $2Conf$, note that OPT incurs no cost as long as it is in the configuration opposite of $2Conf$, but migrating there incurs a transition cost. If OPT remains in the same configuration as $2Conf$, it will incur a cost of $d(S, M) + d(M, S)$ prior to the decision by $2Conf$ to change configurations. Thus, it is always better for OPT to move to the opposite configuration when it finds itself in the same configuration as $2Conf$. In summary, the cost of OPT for $\mathbf{Q'}$ during the first k transitions of $2Conf$ is $(k/2)(d(S, M) + d(M, S))$, which is the same as the transition cost incurred by $2Conf$. Hence, the cost of $2Conf$ on $\mathbf{Q'}$ is at most three times that of OPT. ■

B. N-Configuration Scenario

We extend $2Conf$ to N configurations and describe two heuristics that deal with an exponential number of configurations. In the N-configuration scenario, our $NConf$ algorithm must consider, for each incoming query, all $N - 1$ possible transitions with respect to the current configuration. This requires tracking the cumulative penalty of remaining in the current configuration relative to to *every* alternative and picking the one that benefits query performance the most. Let $x \in \mathbf{C}$ be the current configuration and $y \in \mathbf{C}$ be an alternative in which $y \neq x$. Define $\delta^y_{\max}(k)$ as the maximum cumulative penalty of remaining in x rather than transitioning to y at query q_k (the penalty of remaining in x for q_k is $q_k(x) - q_k(y)$). In the two-configuration scenario, a transition is made when $\delta_{\max}(k)$ exceeds a constant threshold D. In $NConf$, this threshold is no longer constant and is a function of the configuration immediately prior to x and the alternative configuration being considered. Let z be the configuration immediately prior to

x in which the threshold required for transitioning to an alternative configuration y is $d(z,x) + d(x,y)$. The decision to transition to a new configuration by $NConf$ is greedy; that is, $NConf$ transitions to the *first* configuration y that satisfies $\delta^y_{\max}(k) > d(z,x) + d(x,y)$.

$NConf$ is not three-competitive because it transitions immediately to the *first* configuration in which the cumulative penalty exceeds the threshold. This greedy approach is susceptible to oscillations among configurations that exhibit low migration cost, which is a problem in multi-configuration scenarios. In particular, $NConf$ may transition between configurations that yield short-lived benefits with respect to query performance and overlook configurations with a higher cumulative benefit but incurs a large, one-time transition cost. In practice, we expect minor oscillations after $NConf$ finds candidates that perform almost as well as the configuration chosen by OPT because Astronomy workloads do not exhibit rapid changes, such as on an hourly basis.

Performing configuration changes online means that an exhaustive evaluation of every alternative configuration for each incoming query is infeasible (*e.g.* over 51 trillion ways exist to vertically partition a table with 20 attributes). We adopt two heuristics to restrict the search space. First, consider a configuration $x \in \mathbf{C}$, which partitions attributes from one logical relation into separate physical tables. Let each physical grouping of attributes from the same logical relation be denoted as a *fragment*. Thus, no two configuration will have the same set of fragments. Next, we define a neighbor relationship that describes the easiness, in terms of migration cost, of transitioning between configurations. Denote N_x as the set of *neighboring configurations* with respect to x in which a configuration $y \in N_x$ is considered a neighbor of x if transitioning from x to y requires coalescing at most two fragments in x or splitting exactly one fragment of x into two disjoint fragments. Provided x is the current configuration, our heuristic only considers the set of configurations in N_x for transitioning.

Enumeration-based, offline vertical partitioning algorithms [10][18] provide an intuition for neighboring configurations. For instance, AutoPart [10] starts from a configuration in which each attribute is a separate fragment and enumerates candidate configurations by coalescing existing fragments in a pairwise manner. The search space is restricted to small permutations of the current configuration at each iteration, producing candidates that gradually reduce the expected total query execution cost for the workload. This approach to offline vertical partitioning produces configurations that perform well in practice. Neighboring configurations are similar in that each transition changes at most two fragments from the current configuration, which incurs low I/O for each transition. However, restricting the immediate solution to neighbors does not prevent the exploration of the entire space, albeit via small, incremental transitions. Neighboring configurations also provide two desirable properties. First, since the cost of transitioning to a neighbor is relatively low, there is a lower threshold to overcome, which allows $NConf$ to respond quickly to workload changes. Moreover, transitions based on small permutations of the current configuration amortizes the I/O impact of partitioning multiple physical tables so as to limit disruption on the normal operations of a database.

We introduce a pruning heuristic that further reduce the set of neighboring configurations. Workload-based, offline partitioning algorithms [10] invoke the query optimizer to estimate the I/O cost of each query on each hypothetical configuration. Cost estimates for the workload are then used to evaluate candidate configurations. Similarly, to accurately calculate penalty cost in $NConf$, the query optimizer is consulted for each incoming query against every neighboring configuration. Since the set of neighbors is on the order of $O(2^n)$, cost estimation is a major overhead for an online system such as SkyQuery in which query response time is often less than a second. Our approach minimizes invocations of the query optimizer by identifying promising configurations based on groups of attributes that are used frequently in the workload. Cost estimation is only performed for configurations that are expected to have a large impact on query performance.

We describe a pruning heuristic based on the observation that few attribute groupings (on the order of thousands) dominate because Astronomy workloads are template-based [23], which allows us to efficiently filter from a large pool of neighboring configurations. (While many queries are template-based, the set of templates change over time as new templates are introduced and gain importance). This approach is similar to association rules mining [24], which identifies sets of related products in a store based on the purchasing pattern of customers.

To illustrate our pruning heuristic, consider a relation consisting of four attributes, a_1-a_4. Let A_k denote the set of attributes accessed by query q_k. A set of attributes g is an attribute group of q_k if $g \subseteq A_k$. Thus, g indicates a potential grouping which is beneficial; that is, keeping attributes from g in the same physical table lowers the cost of evaluating q_k. Given a query q_k which accesses $\{a_1, a_3, a_4\}$, the pruning heuristic first enumerates all attribute groups of q_k, producing subsets a_1, a_3, a_4, a_1a_3, a_1a_4, a_3a_4 , and $a_1a_3a_4$. We then identify groups that guide transitioning decisions and ignore the rest. Specifically, attribute groups which show that transitioning to a neighboring configuration reduces the cost of evaluating q_k. This includes splitting a fragment to reduce the cost of scanning extraneous attributes or merging fragments to reduce the join cost of selecting data from several attributes.

To illustrate this process, let the current configuration consist of three fragments $\{a_1a_2\}$, $\{a_3\}$, and $\{a_4\}$. The attribute groups of import for q_k are a_3a_4, which supports the coalescing of fragments $\{a_3\}$ and $\{a_4\}$ to reduce join cost, and a_1, which supports the splitting of fragment $\{a_1a_2\}$ to eliminate the cost of scanning attribute a_2. Attribute groups a_3 and a_4 are ignored because they correspond to existing fragments and do not indicate a better alternative. $a_1a_3a_4$ is pruned to avoid double counting because physically manifesting this grouping requires changing more than two fragments from the current configuration (recall that neighbors are formed by coalescing

```
NConf(q_k :query, c :current configuration)
// initially neighbors = {}, prevT = 0
01  u_neighbors = updateWeight(q_k, c)
02  for each n in u_neighbors
03    if n.weight > prevT + d(c,n) and n ∉ neighbors
04      neighbors = neighbors ∪ n
05  for each n in neighbors
06    π(k) = q_k(c) - q_k(n)
07    δ^n_max(k) = max{π(k), δ^n_max(k-1) + π(k)}
08    if δ^n_max(k) > prevT + d(c,n)
09      neighbors = {}, prevT = d(c,n)
10      Transition to n
```

Fig. 6. Online Algorithm for N Configurations

at most two fragments or splitting one existing fragment).

Once attribute groups are evaluated and the neighboring configurations of import are found, weights are assigned to each neighbor that benefits q_k. Weights should capture changes in the relative importance, in terms of expected benefit to query performance, of neighbors as queries evolve over time. $NConf$ uses the I/O cost of evaluating queries against the current configuration for weights so that transitioning to a neighbor with more weight is expected to have a greater impact on query performance than a neighbor with less weight. Thus, if q_k supports a neighbor n, then n's weight increments by $q_k(c)$ in which c is the current configuration. Calculating weights based on the current configuration helps bias optimization efforts toward queries that benefit from re-partitioning. Namely, if a group of queries performs poorly on the current configuration and continues to incur high I/O costs, then neighbors that benefit these queries receive higher weights. In $NConf$, optimization efforts are focused on neighbors that may have accumulated sufficient weight to overcome the threshold for transitioning.

Figure 6 provides the pseudo-code for $NConf$. Lines 1-4 employ the pruning heuristic in which *updateWeight* increments the weights of neighbors that benefit q_k. Neighbors that have accumulated sufficient weights are considered as candidates for transitioning (line 3). The threshold for weights is $prevT + d(c, n)$ in which $prevT$ is the cost of the previous transition and c is the current configuration. Lines 5-10 update the maximum cumulative penalty for each neighbor and transitions to the first neighbor n satisfying the threshold or $\delta^n_{max}(k) > prevT + d(c, n)$.

V. Experiments

In this section, we present an initial evaluation of our online-partitioning algorithm. To prove the validity and generality of our algorithm, we conduct experiments on the TPC-D [25] database and query workload. We are still in the process of evaluating our algorithm in SkyQuery databases. The size of the datasets and the complexity of the workload require a robust framework that we are currently working to establish.

The online partitioning algorithm is evaluated on the 500MB configuration of the TPC-D benchmark [25]. Secondary indices were dropped from the raw database to isolate the benefits of vertical partitioning on query performance. Of the 22 decision support queries from the benchmark, we took a subset of those queries that referenced the *Orders* relation. (These include queries: Q3, Q4, Q5, Q7, Q8, Q9, Q12, and Q13). The *Orders* relation consists of 750,000 rows and eight attributes. One column in particular, *o_comment*, occupies over half of the relation's total size and is accessed only by Q13. Vertical partitioning is performed on the *Orders* table using two 10k query workloads that consist of the eight query types. In the first workload *Wkld_Rand*, queries are generated randomly and each query occurs with equal probability. The second workload *Wkld_Dom* represents a query access pattern that closely matches the SkyQuery workload. In this workload, a few queries dominate with equal probability for a certain length of the query sequence. TPC-D queries Q3, Q4 and Q12 dominate in the initial one-third of the query sequence, queries Q5, Q8, and Q9 dominate for the next one third of the query sequence, and Q13 dominates in the last one-third. The dominance factor is 80%.

We evaluate the *NConf* algorithm against AutoPart [10], a workload-based, offline partitioning tool. The unpartitioned *Orders* relation serves as our base table. We use query response time as the metric for comparing the performance of each approach. To make the comparison fair, we adapt the cost estimation module from AutoPart for evaluating queries against various candidate configurations. The module provides, for a query q_i, an estimate for $q_i(x)$ in which x can be any candidate configuration. AutoPart relies on the query optimizer for estimating the I/O cost of a query by first constructing virtual configurations. The configurations are virtual in that the cost module creates the schema for each candidate configuration but does not populate the configuration with data. To ensure accurate estimates, AutoPart's cost estimation tool supplies the query optimizer with up-to-date system catalog entries and table statistics on the partitioned schema. The statistics are generated from full table scans such that the estimates for virtual configurations track closely with that of real configurations [10]. To account for transition cost, we assume a fixed cost for every transition of 30,000 logical page reads (for 8KB page size). Logical page reads allow us to compare directly with the I/O cost of query execution. We rely on a fixed cost because we are still developing a framework for estimating transition cost. In particular, it is difficult to accurately estimate migration costs that include write overhead and the cost of data definition operations (*i.e. alter table*) for which estimates cannot be obtained from the query optimizer.

All experiments for the 500MB configuration of the TPC-D database ran on Microsoft's SQL Server 2000. Our main workstation is a 3GHz Pentium IV machine with 1GB of main memory and two SATA disks. For performance reasons, we assign logging to one disk and store the database on a second 500GB disk.

Figure 7 shows the average query response time for the online partitioning algorithm compared with AutoPart. On *Wkld_Rand*, AutoPart performs slightly better than the online algorithm. This is due to the warm-up time required by the online algorithm to migrate to the same configuration as that obtained by running AutoPart offline. The online algorithm

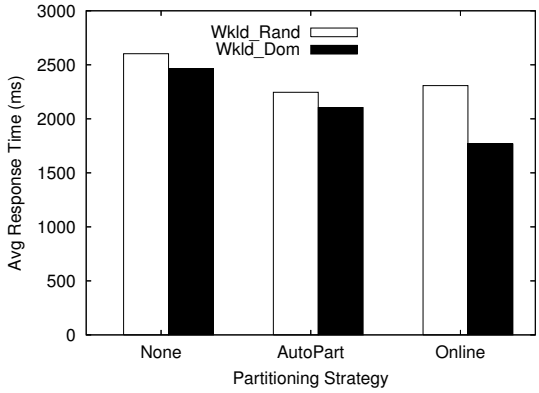

Fig. 7. Query performance by partitioning strategy.

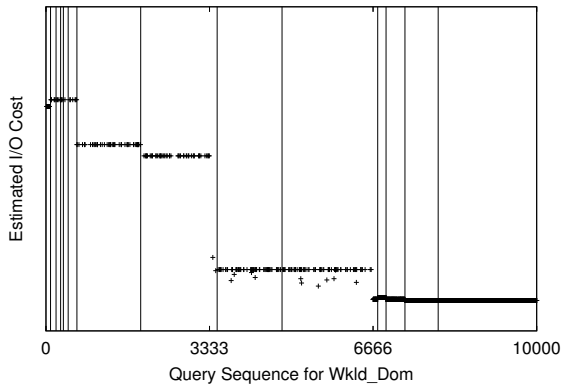

Fig. 8. Estimated I/O Cost for queries in *Wkld_Dom*.

incurs just seven configuration changes prior to arriving at a stable configuration for the remainder of the workload. For *Wkld_Dom*, the online algorithm outperforms AutoPart by 17%. In fact, AutoPart suggests the same configuration for both workloads and is not able to detect changes in the workload sequence. In contrast, our online algorithm made thirteen configuration changes during the course of evaluating *Wkld_Dom*. It adapts the configuration based on queries that dominate the workload sequence.

Figure 8 shows, for the online algorithm, the estimated I/O cost of a subset of queries from *Wkld_Dom* (each data point corresponds to the estimated I/O cost of a query in the workload sequence). In the figure, each vertical line denotes a configuration change. To better illustrate the performance impact before and after each transition, we only plot the I/O cost of Q3 in the first third of the sequence, Q5 in the second third, and Q13 in the final third. Recall that the workload evolves in that queries which dominate the initial third of the workload differs from those that dominate in the second and last third. The online algorithm detects these changes and adapts the configuration accordingly. Seven transitions occur in the initial third in which the configuration oscillates rapidly during the first 500 queries as the algorithm re-groups each attribute in the unpartitioned *Orders* relation. Similarly, in the second and final third of the workload, the algorithm detects the change and makes the appropriate transitions early on. This result illustrates two things: the online algorithm successfully detects changes in the workload and once it adapts to a change, the configuration remains relatively stable thereafter.

The TPC-D benchmark results provide an initial validation of our approach but is not without limitations. First, the impact of caching is ignored in which column insertions and deletions by the caching algorithm may change both workload performance and transition cost. Further, our experiments are limited to a single table with eight attributes, which leave several questions unanswered. While we observe low optimization overhead (only a few dozen configurations are evaluated) on the TPC-D benchmark, it is unclear how the algorithm scales to diverse workloads and larger tables since relations in SDSS can contain several hundred attributes. Also, we assume a fixed migration cost in our experiments and have yet to explore both the proper metric for the cost of migrating between configurations and how this cost can be accurately estimated. Thus, deriving accurate cost estimates with low overhead remains an important focus.

VI. Summary and Future Work

In this paper, we have shown the need for a workload-adaptive physical design solution in Bypass caches. Vertical partitioning is an important technique for physical design as it improves physical design without adding redundant data. Bypass caches receive a continually evolving Astronomy workload, and therefore need an online vertical partitioning technique. We presented online and workload-adaptive algorithms for the vertical partitioning problem that balance improvements in query execution performance with the cost of making physical design changes.

The effectiveness of physical design decisions depends on a good cost estimation module. Cost-estimation in caches should take into account the fact that objects exhibit varying cache resident times. Further, caches are constrained resources, so cost estimation must be efficient but accurate. We plan to integrate fast techniques [26], which cache query plans to improve the efficiency of cost estimation. Currently, these techniques are suited to index selection. While vertical partitioning is an attractive solution for physical design in caches, it may be useful to consider constructing indices for long-lived objects. We plan to study the impact of index selection given that indices will compete for the same cache space. Constructing indices will reduce the response time of queries on objects in the cache but will increase the response time of queries on objects that were evicted because they are bypassed to the backend database. Addressing this trade-off is crucial to the integration of indices.

References

[1] P. Larson, J. Goldstein, H. Guo, and J. Zhou, "MTCache: Mid-Tier Database Caching for SQL Server," in *ICDE*, 2004.

[2] M. Altinel, C. Bornhvd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald, "Cache Tables: Paving the Way for An Adaptive Database Cache," in *VLDB*, 2003.

[3] M. Altinel, Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. G. Lindsay, H. Woo, and L. Brown, "DBCache: Database Caching for Web Application Servers," in *SIGMOD*, 2002.

[4] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson, "Reliability and Security in the CoDeeN Content Distribution Network," in *USENIX*, 2004.

[5] S. Chaudhuri and V. R. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server," in *VLDB*, 1997.

[6] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated Selection of Materialized Views and Indexes in SQL Databases," in *VLDB*, 2000.

[7] S. Agrawal, E. Chu, and V. Narasayya, "Automatic Physical Design Tuning: Workload as a Sequence," in *SIGMOD*, 2006.

[8] T. Malik, R. Burns, and A. Chaudhary, "Bypass Caching: Making Scientific Databases Good Network Citizens," in *ICDE*, 2005.

[9] T. Malik, A. S. Szalay, A. S. Budavri, and A. R. Thakar, "SkyQuery: A Web Service Approach to Federate Databases," in *CIDR*, 2003.

[10] S. Papadomanolakis and A. Ailamaki, "AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning," in *SSDBM*, 2004.

[11] S. Agrawal, V. R. Narasayya, and B. Yang, "Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design," in *SIGMOD*, 2004.

[12] A. Borodin, N. Linial, and M. E. Saks, "An Optimal On-line Algorithm for Metrical Task System," *J. ACM*, vol. 39, no. 4, pp. 745–763, 1992.

[13] A. Szalay, J. Gray, A. Thakar, P. Kuntz, T. Malik, J. Raddick, C. Stoughton, and J. Vandenberg, "The SDSS SkyServer - Public Access to the Sloan Digital Sky Server Data," in *SIGMOD*, 2002.

[14] The Sloan Digital Sky Survey. [Online]. Available: http://www.sdss.org

[15] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, "Vertical Partitioning Algorithms for Database Design," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 680–710, 1984.

[16] N. Bruno and S. Chaudhuri, "An Online Approach to Physical Design Tuning," in *ICDE*, 2007.

[17] The TimesTen Team, "Mid-tier Caching: The TimesTen Approach," in *SIGMOD*, 2002.

[18] M. Hammer and B. Niamir, "A Heuristic Approach to Attribute Partitioning," in *SIGMOD*, 1979.

[19] W. W. Chu and I. T. Ieong, "A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems," *IEEE Trans. Software Eng.*, vol. 19, no. 8, pp. 804–812, 1993.

[20] D. W. Cornell and P. S. Yu, "An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 248–258, 1990.

[21] W. R. Burley and S. Irani, "On Algorithm Design for Metrical Task Systems," in *SODA*, 1995.

[22] W. Bein, L. L. Larmore, and J. Noga, "Uniform Metrical Task Systems with a Limited Number of States," *Inf. Process. Lett.*, vol. 104, no. 4, pp. 123–128, 2007.

[23] X. Wang, T. Malik, R. Burns, S. Papadomanolakis, , and A. Ailamaki, "A Workload-Driven Unit of Cache Replacement for Mid-Tier Database Caching," in *DASFAA*, 2007.

[24] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," in *SIGMOD*, 1993.

[25] TPC-D Benchmark. [Online]. Available: http://www.tpc.org

[26] S. Papadomanolakis, D. Dash, and A. Ailamaki, "Efficient Use of the Query Optimizer for Automated Database Design," in *VLDB*, 2007, pp. 1093–1104.

APPENDIX

Date	Top 2 Queries	Freq	*PhotoPrimary* Attributes
02/08-02/14 (17k queries)	SELECT '' + cast(p.objId as varchar(20)) + '' as objID, p.run, p.rerun, p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u, p.g, p.r, p.i, p.z, p.Err_u, p.Err_g, p.Err_r, p.Err_i, p.Err_z FROM fGetNearbyObjEq(185,-0.5,3) n, PhotoPrimary p WHERE n.objID=p.objID	5585	camcol, cx, cy, cz, dec, err_g, err_i, err_r, err_u, err_z, field, g, htmid, i, obj, objid, r, ra, rerun, run, type, u, z
	SELECT distinct p.run, p.rerun, p.camcol, p.field FROM fGetNearbyObjEq(77.699896,64.913318,15.0) n, PhotoPrimary p WHERE n.objID=p.objID	1864	
02/15-02/21 (19k queries)	SELECT p.objID, rc.name, s.name, p.ra, p.dec, ph.name, p.u, p.g, p.r, p.i, p.z, o.distance FROM ((PhotoPrimary p inner join PhotoType ph on p.type = ph.value) left join RC3 rc on p.objid = rc.objid) left join Stetson s on p.objid = s.objid), dbo.fGetNearbyObjEq(18.87837,-0.86083,0.5) o WHERE o.objid = p.objid and p.type = ph.value order by o.distance	7229	camcol, ***colv,*** ***colverr,*** cx, dec, ***extinction_g,*** ***extinction_i,*** ***extinction_r,*** ***extinction_u,*** ***extinction_z,*** field, ***flags,*** g, i, ***isoa_i,*** ***isob_i,*** ***isophi_i,*** ***lnlstar_g,*** ***lnlstar_i,*** ***lnlstar_r,*** ***lnlstar_u,*** ***lnlstar_z,*** ***mcr4_i, mel_i,*** ***me2_i, mrrcc_i,*** obj, objid, ***primtarget,*** ***probpsf,*** ***psfmag_g,*** ***psfmag_i,*** ***psfmag_r,*** ***psfmag_u,*** ***psfmag_z,*** ***psfmagerr_g,*** ***psfmagerr_i,*** ***psfmagerr_r,*** ***psfmagerr_u,*** ***psfmagerr_z,*** r, ra, rerun, ***rowv,*** ***rowverr,*** run, ***status,texture_g, texture_i,*** ***texture_r,*** ***texture_u,*** ***texture_z,*** type, u, z
	SELECT ra, dec, type, flags, status, primTarget, probPSF, run, rerun, camcol, field, obj, psfMag_u, extinction_u, psfMagErr_u, psfMag_g, extinction_g, psfMagErr_g, psfMag_r, extinction_r, psfMagErr_r, psfMag_i, extinction_i, psfMagErr_i, psfMag_z, extinction_z, psfMagErr_z, texture_u, texture_g, texture_r, texture_i, texture_z, lnLStar_u, lnLStar_g, lnLStar_r, lnLStar_i, lnLStar_z, mE1_i, ME2_i, mRrCc_i, mCr4_i, isoA_i, isoB_i, isoPhi_i, rowv, rowvErr, colv, colvErr FROM PhotoPrimary WHERE (type = 6) and (ra >= 160.000000 and ra < 161.000000) and (dec >= -2.000000 and dec < -1.000000)	2259	
02/22-02/28 (19k queries)	SELECT '' + cast(p.objId as varchar(20)) + '' as objID, p.run, p.rerun, p.camcol, p.field, p.obj, p.type, p.ra, p.dec, p.u, p.g, p.r, p.i, p.z, p.Err_u, p.Err_g, p.Err_r, p.Err_i, p.Err_z FROM fGetNearbyObjEq(185,-0.5,3) n, PhotoPrimary p WHERE n.objID=p.objID	5460	camcol, cx, cy, cz, dec, err_g, err_i, err_r, err_u, err_z, extinction_g, extinction_i, extinction_r, extinction_u, extinction_z, field, flags, g, htmid, i, ***modelmag_g,*** ***modelmag_i,*** ***modelmag_r,*** ***modelmag_u,*** ***modelmag_z,*** ***modelmagerr_g,*** ***modelmagerr_i,*** ***modelmagerr_r,*** ***modelmagerr_u,*** ***modelmagerr_z,*** obj, objid, primtarget, probpsf, r, ra, rerun, run, status, type, u, z
	SELECT ra, dec, type, flags, status, primTarget, probPSF, run, rerun, camcol, field, obj, modelMag_u, extinction_u, modelMagErr_u, modelMag_g, extinction_g, modelMagErr_g, modelMag_r, extinction_r, modelMagErr_r, modelMag_i, extinction_i, modelMagErr_i, modelMag_z, extinction_z, modelMagErr_z FROM PhotoPrimary WHERE (type = 3) and (ra >= -0.737262 and ra < 1.262757) and (dec >= -0.750681 and dec < 1.249319)	4279	

Fig. 9. Top two most frequent query types during each week. The right-most column lists attributes from the *PhotoPrimary* relation that are accessed by these queries.