Apollo: Learning Query Correlations for Predictive Caching in Geo-Distributed Systems

Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Scott Foggo and Anil Pacaci

E-mail:{bjglasbe, mtabebe, kdaudjee, sjfoggo, apacaci}@uwaterloo.ca
David R. Cheriton School of Computer Science, University of Waterloo
Submission Type: Research

ABSTRACT

The performance of modern geo-distributed database applications is increasingly dependent on remote access latencies. Systems that cache query results to bring data closer to clients are gaining popularity, but they do not dynamically learn and exploit access patterns in client workloads. We present a novel prediction framework that identifies and makes use of workload characteristics obtained from data access patterns to exploit query relationships within an application’s database workload. We have designed and implemented this framework as Apollo, a system that learns query patterns and adaptively uses them to predict future queries and cache their results. Through extensive experimentation with two different benchmarks, we show that Apollo provides significant performance gains over popular caching solutions through reduced query response time. Our experiments demonstrate Apollo’s robustness and scalability as a predictive cache for geo-distributed database applications.

1 INTRODUCTION

Modern distributed database systems and applications frequently have to handle large query processing latencies resulting from the geo-distribution of data [10, 12, 40]. Industry reports indicate that even small increases in client latency can result in significant drops in both web traffic [20] and sales [3, 30]. A common solution to this latency problem is to place data closer to clients [36, 37] using caches, thereby avoiding costly remote round-trips to datacenters [27]. Static data, such as images and video content, is often cached on servers geographically closer to clients than the datacenters. These caching servers, called edge nodes, are a crucial component in industry architectures. To illustrate this, consider Google’s datacenter and edge node locations in Figure 1. Google has comparatively few datacenter locations relative to edge nodes, and the latency between the edge nodes and datacenter can be quite large. Efficiently caching data on these edge nodes substantially reduces request latency for clients.

Figure 1: Google’s datacenter and edge node locations.

Figure 2: A set of motivating queries in TPC-W’s Order Display web interaction. Boxes of the same color indicate shared values across queries.

Existing cache solutions for edge nodes and content delivery networks (CDN) focus largely on static data, therefore necessitating round trips to remote data centers for requests relying on dynamic data [21]. Since the majority of webpages today are generated dynamically [5], a large number of requests bypass the caches and incur heavy latency penalties. We address this concern in Apollo, a system that exploits client access patterns to intelligently prefetch and cache dynamic data on edge nodes.

Database client workloads often exhibit query patterns, corresponding to application usage patterns. In many workloads [1, 9, 41], queries are highly correlated. That is, the execution of one query determines which query executes next and with what parameters. These dependencies provide opportunities for optimization through predictively caching queries. In this paper, we focus on discovering relationships among queries in a workload. We exploit the discovered relationships to predictively execute future dependent queries. Our focus is to reduce the response time of consequent queries by predicting and executing them, caching query results ahead of time. In doing so, clients can avoid contacting a database located at a distant datacenter, satisfying queries instead from the cache on a closer edge node.

As examples of query patterns, we consider a set of queries from the TPC-W [41] benchmark. In this benchmark’s Order Display web interaction, shown in Figure 2, we observe that the second query is dependent upon the result set of the first query. Therefore, given the result set of the first query, we can determine the input set of the second query, predictively execute it, and cache its results. After the second query has executed, we can use its result set as input to the third query, again presenting an opportunity for predictive caching. Similar scenarios abound in the TPC-W and TPC-C benchmarks, such as in the TPC-W Best-Seller web interaction and in the TPC-C Stock level transaction. Examples that benefit from such optimization, including real-world customer scenarios, have been previously described [9].
A client’s database workload is comprised of a stream of queries and the transitions between them. These queries are synthesized into the query transition graph, which is at the core of our predictive framework. From this query transition graph, we discover query relationships, dependencies and workload characteristics for use in our predictive framework. The predictive framework stores query result sets in a shared local cache system, querying the remote database if a client submits a query for which the cache does not have the results.

Figure 3 gives a high level overview of how incoming queries are executed, synthesized into the query transition graph, and used for future query predictions. Incoming queries are routed to the query processor, which retrieves query results from a shared local query result cache, falling back to a remote database on a cache miss. Query results are immediately returned to the client and, together with their source queries, are mapped to more generalized query template representations (Section 2.1). These query templates are placed into per-client queues of queries called query streams, which are continuously scanned for relationships among executed queries. Query relationships are synthesized into the query transition graph and then used to detect query correlations, discovering dependencies among executed queries and storing them in a dependency graph. This dependency graph is used by the prediction engine to predict subsequent queries given client queries that have executed.

Although we focus on geographically distributed edge nodes with remote datacenters, Apollo can also be deployed locally as a middleware cache. Our experiments in Section 4 show that either deployment environment can benefit significantly.

Next, we discuss the abstractions and algorithms of our predictive framework, describing how queries flowing through the system are merged into the underlying models and used to predict future queries.

2 PREDICTING QUERIES

A client’s database workload is comprised of a stream of queries and the transitions between them. These queries are synthesized into the query transition graph, which is at the core of our predictive framework. From this query transition graph, we discover query relationships, dependencies and workload characteristics for use in our predictive framework. The predictive framework stores query result sets in a shared local cache system, querying the remote database if a client submits a query for which the cache does not have the results.

Figure 3 gives a high level overview of how incoming queries are executed, synthesized into the query transition graph, and used for future query predictions. Incoming queries are routed to the query processor, which retrieves query results from a shared local query result cache, falling back to a remote database on a cache miss. Query results are immediately returned to the client and, together with their source queries, are mapped to more generalized query template representations (Section 2.1). These query templates are placed into per-client queues of queries called query streams, which are continuously scanned for relationships among executed queries. Query relationships are synthesized into the query transition graph and then used to detect query correlations, discovering dependencies among executed queries and storing them in a dependency graph. This dependency graph is used by the prediction engine to predict consequent queries given client queries that have executed.

Although we focus on geographically distributed edge nodes with remote datacenters, Apollo can also be deployed locally as a middleware cache. Our experiments in Section 4 show that either deployment environment can benefit significantly.

Next, we discuss the abstractions and algorithms of our predictive framework, describing how queries flowing through the system are merged into the underlying models and used to predict future queries.

2.1 Query Templates

Using a transition graph to reason about query relationships requires a mapping from database workloads (queries and query relationships) to transition structures (query templates and template transitions). We propose a formalization of this mapping through precise definitions, and then show how our model can be used to predict future queries.

Queries within a workload are often correlated directly through parameter sharing. Motivated by the Stock Level transaction in the TPC-C benchmark, consider an example of parameter sharing in which an application executes query Qt-1 to look up a product id followed by query Qt to check the stock level of a given product id. A common usage pattern is to execute Qt-1 and then use the returned product id as an input to Qt to check that product’s stock level. In this case, Qt is directly related to Qt-1 via a dependency relationship. Specifically, Qt relies on the output of Qt-1 to execute. We generalize our model by tracking relationships among query templates instead of parameterized queries. Two queries Qt-1 and Qt have the same query template if they share the same statement text barring constants that could logically be replaced by placeholders for parameters values (‘?’). Each query template is represented by a node in the query transition graph.

Below is an example of two queries (Qt-1, Qt) and their corresponding templates (Qt-1’, Qt’):

\[ Q_{t-1}: \text{SELECT C_ID FROM CUSTOMER WHERE C_UNAME = 'Bob' and C_PASSWD = 'pwd'} \]
\[ Q_t: \text{SELECT C_ID FROM CUSTOMER WHERE C_UNAME = 'pwd' and C_PASSWD = 'pwd2'} \]

\[ Q_{t-1}': \text{SELECT C_ID FROM CUSTOMER WHERE C_UNAME = 'Alice' and C_PASSWD = 'pwd2'} \]
\[ Q_t': \text{SELECT C_ID FROM CUSTOMER WHERE C_UNAME = 'pwd2'} \]
Note that although the above two original queries differ, their query templates are the same. Therefore, a node’s transitions in the transition graph are based on query relationships from both \( Q_1 \) and \( Q'_1 \).

2.2 Query Template Relationships

To find query template relationships, we implement the transition graph as a frequency-based Markov graph constructed via online learning. We exploit the memory-less property of Markov models to simplify transition probability calculations — transition probabilities are based solely on the previous query the client executed.

We monitor incoming queries, map them to query templates and calculate transition probabilities. In particular, for any two templates \( Q_{t_1}, Q_{t_2} \), we create an edge from \( Q_{t_1} \) to \( Q_{t_2} \) if \( Q_{t_1} \) is executed after \( Q_{t_2} \). We store the probability of \( Q_{t_1} \) executing after \( Q_{t_2} \) on this edge, and refer to it as \( P(Q_{t_1}|Q_{t_2}) \). If this probability is larger than some configurable threshold \( \tau \), we say \( Q_{t_1} \) is related to \( Q_{t_2} \).

The \( \tau \) parameter serves as a configurable confidence level threshold for query template relationships. More concretely, the \( \tau \) parameter provides the minimum required probability for \( Q_{t_1} \) executing after \( Q_{t_2} \) to infer that they are related. By choosing \( \tau \) appropriately, we can limit the predictive queries executed after seeing \( Q_{t_1} \) to only those that are highly correlated to it. In doing so, we ensure that our predictions have a high degree of accuracy and avoid inundating the database with predictive executions of unpopular queries.

\[
P(Q_{t_1}|Q_{t_2}) \text{ is too broad to capture only fine-grained query template relationships. Given enough time, almost all of the query templates in a workload could be considered related under the above definition. From a semantic perspective, two templates should not be considered related if there is a significant time gap between them, thus motivating a temporal constraint. Furthermore, by placing a temporal restriction on the relationship property, we reduce the time needed to look for incoming related templates. Consequently, we define a configurable duration, } \Delta t, \text{ which specifies the maximum allowable time separation between related query templates.}
\]

**Definition 2.1.** For any two query templates \( Q_{t_1}, Q_{t_2} \), in which \( Q_{t_2} \) is executed \( T \) at most time units apart from \( Q_{t_1} \), if \( P(Q_{t_1}|Q_{t_2}; T \leq \Delta t) > \tau \) for some threshold parameter \( \tau \in [0, 1] \), we say \( Q_{t_1} \) is a related query template of \( Q_{t_2} \).

To learn a transition graph representing \( P(Q_{t_1}|Q_{t_2}; T \leq \Delta t) \), we map executed queries to query templates and place them at the end of per-client queues called query streams. Since each client has its own stream and transition graph, we avoid expensive lock contention when updating the graphs and computing transition probabilities.

Algorithm 1 runs continuously over client query streams, updating their corresponding transition graphs. Intuitively, the algorithm scans the query stream, looking for other query templates that executed within \( \Delta t \) of the first query template, adding edge counts to their corresponding transition graph nodes, afterwards incrementing the vertex count indicating number of times the template has been seen. To calculate the probability of \( P(Q_{t_1}|Q_{t_2}; T \leq \Delta t) \), we take the edge count from \( Q_{t_1} \) to \( Q_{t_2} \) and divide by the vertex count for \( Q_{t_1} \). To use the variables directly from Algorithm 1, the probability we execute a query template \( Q_{t_1} \) within \( \Delta t \) of a query template \( Q_{t_2} \) is given by \( \frac{w_e(Q_{t_1}, Q_{t_2})}{w_v(Q_{t_1})} \). Per Definition 2.1, if this probability exceeds \( \tau \), then \( Q_{t_1} \) is considered related to \( Q_{t_2} \).

The choice of the \( \Delta t \) parameter can impact prediction efficacy. If \( \Delta t \) is too high, it is possible that relationships will be predicted where there are none; if \( \Delta t \) is too low, we may not discover relationships where they are present. Although the choice of \( \Delta t \) is workload dependent, there are some indicators (such as query arrival rate) that aid us in choosing an appropriate value. If \( P(Q_{t_1}|Q_{t_2}; T \leq \Delta t) \) is high for a fixed \( Q_{t_1} \) and many different \( Q_{t_2} \), then either \( Q_{t_1} \) is a common query template with many quick-executing related query templates, or \( \Delta t \) is set too high. If this holds for many different \( Q_{t_1} \), then \( \Delta t \) can be decreased. A similar argument holds for decreasing \( \Delta t \). We discuss selection of \( \Delta t \) and \( \tau \) values for various workloads in Section 4.7.

A key property of our model is that it uses online learning to adapt to changing workloads. As new query templates are observed, query template execution frequencies change, or query relationships adjust the transition graph adjusts to learn the changed workload. Moreover, online learning precludes the need to undergo expensive offline training before deployment. Instead, our model rapidly learns client-specific workloads and takes action immediately.

2.3 Parameter Mappings

Predictive query execution requires a stronger relationship between queries than the transition graph provides. In addition to
queries being related, they must also exhibit a dependency relationship.

To provide predictive execution capabilities, we record the output sets of query templates and match them with the input sets of templates that we have determined are related based on the transition graph. We then confirm each output column to input variable mapping over a verification period, after which only the mappings present in every execution are returned.

As a concrete example, consider the TPC-W queries from Figure 2. We will refer to the query template for the first query as $Q_1$, and the second query as $Q_2$. In the first stage of tracking, we observe which query templates have executed within $\Delta t$ of $Q_1$. Once $Q_1$ has executed enough times (according to the verification period), we begin to construct mappings among the query templates. After $Q_1$ finishes an execution, we record its output set. When any of $Q_1$’s related query templates (in this case assume only $Q_2$) are executed, we record their input sets. We then check if any column’s result in $Q_2$’s output set maps to the input parameters of $Q_2$. If they do, we record the matching output columns with their corresponding input argument position. If the same mappings are observed across the verification period, we infer that these mappings always hold.⁴ If a query template has mappings for every one of its input arguments from a set of prior templates, we can predict a query by forwarding parameters from its prior template’s result sets as soon as they are available. In this case, we say the query template is a candidate for predictive execution given its prior query templates’ result sets. Similarly, we discover mappings between $Q_2$ and $Q_1$ and use them to execute $Q_3$ given $Q_2$’s result set.

2.4 Pipelining Query Predictions

Parameter mappings among query templates enable predictive execution of queries as soon as their input sets are available via prior template execution. In the previous section, we showed how prior query result sets can be used with template mappings to predict future queries. However, it may be the case that the prior query templates are also predictable, forming a hierarchical tree of dependencies among templates. We exploit these relationships by pipelining query predictions. Pipelining uses result sets from predictively executed queries as input parameters for future predictions, thereby enabling predictions several steps in advance.

Figure 4: An example of pipelining within a dependency hierarchy. The arrows represent a mapping from a prior query template’s output set to the consequent query template’s input set.

Figure 4 illustrates how pipelining can be used to form extended chains of predictive executions using the TPC-W example from Figure 2. $Q_1$ has a mapping to $Q_2$, which in turn has a mapping to $Q_3$. If $Q_1$ is executed, we can forward the result set as input to predictively execute $Q_2$. Once $Q_2$ has also been executed, we can predictively execute $Q_3$. As such, $Q_2$ is fully defined given the result set of $Q_1$, and $Q_3$ is fully defined given the result set of $Q_2$.

We formalize the notion of fully defined queries:

**Definition 2.2.** A fully defined query template (FDQ) $Q_j$ has all of its inputs provided by some set, possibly empty, of prior query templates $Q_{t_1}, Q_{t_2}, \ldots, Q_{t_m}$ and each $Q_{t_m}$ ($\forall m \in [1,k]$) is either:

1. a fully defined query template
2. a dependency query template, required to execute $Q_j$.

Per Definition 2.2, both $Q_2$ and $Q_3$ are FDQs, but $Q_1$ is simply a dependency query. These definitions elucidate the dependency graph nature of FDQs — each node in this graph corresponds to a query template, with inbound and outbound edges corresponding to inbound and outbound parameter mappings, respectively. The transition graph induces the dependency graph, but is stored and tracked separately. By keeping the dependency graph separate, we reduce contention on it. Once the dependency graph matches the current workload, it will not need to be modified until the workload changes.

**Algorithm 2 Core Prediction Algorithm**

**Input:** executed query template $Q_t$

**record_query_template($Q_t$)**

**new_fdqs = find_new_fdqs($Q_t$)**

**rdy_fdqs = mark_ready_dependency($Q_t$)**

**rdy_fdqs = rdy_fdqs $\cup$ new_fdqs**

**ordered_fdqs = find_allRunnable_fdqs(rdy_fdqs)**

**for all** ready_fdq $\in$ ordered_fdqs **do**

**execute_fdq(ready_fdq)**

**end for**

Discovering new FDQs, managing FDQ dependencies, and pipelining predictions comprise the main routine of the predictive framework. Upon execution of a client-provided instance of query template $Q_t$, the engine executes the loop shown in Algorithm 2. The engine records $Q_t$’s result set and input parameters in the query transition graph (Section 2.3), looks for parameter mappings, and records found dependencies in the dependency graph. This query template is then marked as executed so that FDQ pipelines that depend on its result can proceed. Any queries that are determined ready for execution given the result of this query (and previously executed queries) are then executed, feeding parameters from their dependent queries’ result sets as necessary. The dependencies are then reset, waiting for future invocations with which to predict queries. The dependency graph is stored as a hash map with edges between dependent queries, allowing Apollo to quickly determine which FDQs are ready for execution given an executed query.

Always defined query templates (ADQs) are a subset of FDQs, requiring that all of their prior query templates are FDQs. They comprise an important subclass of fully defined queries since their dependencies are always satisfied; they can be executed and cached at any time. As a concrete example, “SELECT COUNT(*) FROM shopping_cart” is an ADQ because all of its input parameters (the empty set) are always satisfied.

It follows from Definition 2.2 that an FDQ is an ADQ if and only if all of its inputs are provided by ADQs. Consequently, ADQ

---

⁴If future executions disprove a mapping, we will mark that mapping invalid and preclude the template from predictive execution if its dependencies are no longer met.
hierarchies are discovered by recursively checking the dependency structure of the FDQ.

3 APOLLO

In this section, we present Apollo, a system that implements the predictive framework described in Section 2. Apollo is a system layer placed between a client application and the database server. Application clients submit queries to the Apollo system, which then interacts with the database system and cache to return query results.

Apollo uses Memcached [19], a popular industrial-strength distributed caching system, as the query result cache. Every executed query has its result set placed in Memcached, which employs the popular Least Recently Used (LRU) eviction policy. With predictive caching enabled, Apollo also places predictively executed query results into the cache, increasing the number of cache hits and thereby the overall system performance. Apollo’s predictive engine operates in a strictly complementary manner where queries are passed through to the database and cache unchanged, keeping the effective workload behaviour the same. Apollo executes predicted queries and caches them a priori, reducing response times through correlated query result caching.

Since Apollo is implemented in the Java programming language, we use the JDBC API to submit queries to the remote MySQL database. The JDBC API makes Apollo database agnostic and therefore portable, allowing MySQL to be easily swapped for any other JDBC compliant relational database system if desired.

To efficiently track query templates within Apollo, we identify queries based on a hash of their constant independent parse tree. A background thread processes the SQL query strings placed into the query stream, parsing and then hashing them into a 64-bit identifier. All parameterizable constants are replaced by a fixed string, and therefore share the same hash code. Thus, queries with the same text modulo parameterizable constants have the same hash.

Hashes can be computed very efficiently and are used internally to refer to query templates. Apollo uses them to look up nodes in the transition graph, and to find statistics and parameters we have stored for each query template. Hash collisions are very rare due to the length of the hash and common structures that SQL statements share. Due to the strictly complementary nature of Apollo, query template hash collisions are guaranteed not to introduce incorrect system behaviour.

3.1 Prediction Engine

When a client submits a query, it has its results retrieved from the local cache or executed against the remote database, then placed into Apollo’s query stream and evaluated by the prediction engine. Background threads use the query stream to construct the transition graph described in Section 2, processing executed queries into query templates. The core prediction routine from Section 2.4 is then invoked: new FDQs are discovered from the underlying transition graph, the dependency graph is updated, and future queries are predicted using pipelining. We now detail each of these subroutines, showing how these operations are carried out efficiently.

Algorithm 3 shows how new FDQs are discovered. First, the transition graph is consulted for all related query templates (templates with inbound edges from \( Q_i \)) since these are the templates that may have new mappings from \( Q_i \)’s result set. \( Q_i \) itself is also checked, since it may be an ADQ (if it has no input parameters). For each query template \( Q_i \) that has no recorded dependency information in the dependency graph, the transition graph is checked to see which templates have mappings to them. If each of \( Q_i \)’s input parameters are satisfied by its prior query templates, then by Definition 2.2 we know that it is an FDQ. An FDQ structure is constructed for \( Q_i \) and its dependencies are recorded in the dependency graph. For efficiency, we represent the dependency graph as a hashmap from dependency query templates to dependent templates and their full dependency list. Therefore, determining newly satisfied FDQs can be performed quickly with simple lookup operations.

**Algorithm 3 find_new_fdqs**

**Input:** a query template \( Q_i \)

**Output:** a set of newly discovered FDQs

\[
\text{queries_to_check} = \text{get_related_queries}(Q_i) \\
\text{queries_to_check} = \text{queries_to_check} \cup \{ Q_i \} \\
\text{new_fdqs} = \{ \} \\
\text{for all } Q_j \in \text{queries_to_check} \text{ do} \\
\quad \text{if } \text{already_seen_deps}(Q_j) \text{ then} \\
\quad \quad \text{p_mappings} = \text{get_prior_query_mappings}(Q_j) \\
\quad \quad \text{if } \text{have_enough_mappings}(Q_i, p_mappings) \text{ then} \\
\quad \quad \quad \text{fdq} = \text{construct_fdq}(Q_i, p_mappings) \\
\quad \quad \quad \text{unresolved_deps} = \text{get_dependencies}(fdq) \\
\quad \quad \quad \text{add_to_dep_graph}(unresolved_deps, fdq) \\
\quad \quad \quad \text{mark_seen_deps}(fdq) \\
\quad \quad \text{new_fdqs} = \text{new_fdqs} \cup \{ fdq \} \\
\quad \text{end if} \\
\text{end if} \\
\text{end for} \\
\text{return new_fdqs}
\]

Apollo ensures that there exists only one instance of an FDQ hierarchy throughout the system so that mapping updates affect both the FDQ and any FDQ structures it is a child of. To do so, we track the FDQs that the system has constructed before, returning previously constructed FDQ if applicable. During FDQ construction, dependency loops are detected and returned as dependency queries in an FDQ hierarchy. If all children of an FDQ are tagged as ADQs, or if an FDQ has no parameters and no children, then it is tagged as an ADQ and stored for use during cache reload (Section 3.4.2). Dependency queries are marked as unresolved dependencies on the FDQ and used to determine when an FDQ is ready for execution. Algorithm 4 shows how dependencies for known FDQs are tracked and used for predictive execution. For each FDQ that may be ready for execution given \( Q_i \), we mark \( Q_i \) as executed in its dependency list. If all of its dependencies are now satisfied, we add it to a list of “ready FDQs”, resetting its dependencies so that they must be satisfied again before we determine the FDQ as being ready for future execution.

Algorithm 4 is used as part of a breadth-first approach to determine all runnable FDQs given the current query state. First, Apollo determines which FDQs are executable given the current system state and a newly executed query, adding them (as previously mentioned) to the list of ready FDQs. Apollo then determines which other FDQs are executable given this FDQ list, repeating...
Write queries are never predictively executed (to prevent unnecessary rollbacks) and always execute against the database. The client has a single transition graph. However, by maintaining multiple independent transition graphs with different Δt intervals, we are able to determine the likelihood of a given query being executed by the client in each of these windows. We use this technique to predict if a client will retrieve the results for a predicted query before its session is increased (and consequently the query’s results become stale). Apollo will predictively execute and cache only queries that it deems are likely to be used.

Since Apollo handles many concurrent clients, multiple clients may simultaneously try to execute the same read query. In these cases, it is beneficial to execute the query only once and return its result set to the waiting clients. Optimizing these queries is particularly important for predictive execution, since a predicted query may not have finished execution before a client requests its result set.

Before executing a query, Apollo consults a hashtable to determine if it is already executing. If so, it will block, waiting until the query returns and passing along its result set. If not, it will record an entry in the hashtable with a semaphore for other clients and predictive pipelines to wait on. In this way, only one copy of a read query is executing at once, including shared predictive query pipelines for multiple clients.

To maintain client session consistency, Apollo records the version vector of the client executing the query in the hashtable with the semaphore. Any other client with an earlier version vector that wishes to execute the same query concurrently can subscribe to its results and update its version vector to match.

When Apollo determines that a client’s query has multiple usable versions of its results cached, Apollo will always use the earliest version, regardless of whether another usable version is already being retrieved for a different client. Experimentally, we determined that it is better to retrieve results for earlier versions, since later versions will update the client and may therefore miss on other cache hits. Similarly, if Apollo must retrieve the result set from the database, Apollo will always subscribe to any ongoing database retrievals of the same query.

## 3.4 Session-Aware Caching

Since write queries increment client version vectors, they preclude the client from reading any previously cached values. Therefore, if a client executes a write query after a predictive query is issued on that client’s behalf, the predicted query results may be stale and unusable. If so, the system will have performed unnecessary work to execute and cache the query. To minimize the effects of writes on system performance, we avoid predictively executing queries whose results are likely to become stale before client queries can use their results (Section 3.4.1). Since ADQs can be executed at any time, we keep valuable ADQs in the cache by reloading them if their results become outdated (Section 3.4.2).

### 3.4.1 Preventing Unusable Predictions

Apollo determines the likelihood of a write query or cache miss occurring using the query transition graph. Previously, we have mentioned that each client has a single transition graph. However, by maintaining multiple independent transition graphs with different Δt intervals, we are able to determine the likelihood of a given query being executed by the client in each of these windows. We use this technique to predict if a client will retrieve the results for a predicted query before its session is increased (and consequently the query’s results become stale). Apollo will predictively execute and cache only queries that it deems are likely to be used.

To determine if predictively executing and caching a query will be helpful, Apollo first estimates the time it will take for the query to be executed and cached. Since all predictable queries are by definition FDQs, we use a simple estimate: the time to predictively
execute an FDQ is given by the time it will take to execute its dependencies (if necessary) and the time to execute the FDQ itself. We calculate this estimate recursively: for a target FDQ, we return the maximum time to execute its dependency queries and add on the time needed to execute the FDQ. In essence, this process returns the longest expected path from the child weighted by average query runtimes. To provide an approximation of individual query runtimes, we use a simple average for each query template. Although more sophisticated methods can be used [4, 43] to estimate query runtimes, we found that this method yields enough accuracy to determine the runtime of predicted query while still being performant.

Once the runtime for the given FDQ \( f \) has been determined (say \( t \)), Apollo looks up the client’s transition graph with largest interval \( \Delta t \) where \( \Delta t < t \). It then uses this graph to determine the likelihood of the client executing a query that would cause \( f \)’s results — or the results of its dependencies — to become stale while \( f \) is executing. If this likelihood is sufficiently high (given the \( \tau \) threshold), we avoid executing \( f \) to save on computation costs. Therefore, only queries that are likely to be executed and useful to clients are predictively cached.

Although increasing the number of transition graphs per client necessitates additional processing of the query stream, we find that the simplicity of the query transition graph construction algorithm (Algorithm 1) combined with a configurable (but small) number of models per client results in low computational overhead for the system. Furthermore, since workloads [1, 41] tend to have a small number of unique query templates, the storage overhead is minimal.

3.4.2 Informed ADQ Reload. Write queries update the client version vector, and therefore provide an opportunity for optimization through informed query result reload. As ADQ dependencies are always satisfied and can be executed at any time, we immediately reload valuable ADQ hierarchies after a client executes a write query. Since there can be many ADQs and reloading a hierarchy may be computationally expensive, we limit ADQ reload to only those predictions for query templates considered valuable according to the cost function \( \text{cost}(Q_t) = P(Q_t) \cdot \text{avg}_t \text{rt}(Q_t) \).

Specifically, the estimated cost of an ADQ on the system is given by the probability of the ADQ executing and the estimated ADQ runtime. If the cost of the ADQ exceeds a predefined threshold \( \alpha \), we reload it into the cache. We discuss \( \alpha \) and its effects further in Section 4.7.

4 PERFORMANCE EVALUATION

In this section, we present the system setup used to conduct experiments followed by performance results. Apollo is compared against Memcached [19], a popular mid-tier cache used in database/storage systems, as well as the Fido predictive cache [32]. We compare these systems using average query response time and tail latencies, which have been observed to contribute significantly to user experience and indicate concurrent interaction responsiveness [28].

The Fido engine serves as a drop-in replacement for Apollo’s prediction engine, and uses Palmer et al.’s associative-memory technique [32] for query prediction, scanning client query streams to predict upcoming queries. Fido’s approach is fundamental such that it has been popularly employed to prefetch webpages in content delivery networks [39], objects in object oriented databases [7], and preloading of documents onto client devices [18]. Its widespread use and implementation-independent middleware prediction engine make it particularly well suited as a comparison point against Apollo.

The remainder of this section is organized as follows. Section 4.1 describes our experiments’ setup and Section 4.2 provides performance experiments for TPC-W. In Section 4.3, we use TPC-C to assess Apollo’s scalability under increasing client load. Section 4.4 showcases Apollo’s ability to adapt changing workloads using online learning. Geographic latency experiments and multi-Apollo instance experiments are shown in sections 4.5 and 4.6 respectively, and Section 4.7 presents a sensitivity analysis of Apollo’s configurable parameters.

4.1 Experimental Setup

Our experiments use a geo-distributed setup in which Amazon EC2 nodes are located in the US-East (N. Virginia) region for: (i) Apollo with 16 virtual CPUs, 64 GB of RAM and a 50 GB SSD (ii) Memcached on a machine with 2 virtual CPUs, 4 GB of RAM, and a 50 GB SSD (iii) a node with concurrent clients running our benchmarks with 16 virtual CPUs, 64 GB of RAM, and a 50 GB SSD. We deploy a database machine in the US-West (Oregon) region for our experiments, which has 16 virtual CPUs, 64 GB of RAM, a 250 GB SSD, and uses MySQL v5.6 as the database. For each experiment, Memcached uses a cache size 5% of the size of the remote database to demonstrate that Apollo is effective with limited cache space. Our results are averages computed over at least 5 independent runs with 95% confidence intervals shown as bars around the means.

Our experiments have three primary configurations: the Memcached configuration (in which the cache has been warmed for 20 minutes prior to benchmarking), the Apollo caching configuration, and a Fido prediction engine configuration [32]. In the Memcached configuration, we check for query results in the cache and forward queries on cache misses to the remote database, caching the retrieved query results. The Apollo and Fido configurations have the same behaviour, but Apollo additionally uses the predictive framework from Section 2 and Fido uses its own predictive engine, which is detailed below.

Unlike Apollo, Fido functions at an individual query level rather than query templates. More concretely, if queries \( Q_1, Q_2, \ldots, Q_n \) are present in a client’s query stream, Fido looks for a stored pattern that is prefixed by them, say \( Q_1, Q_2, \ldots, Q_a, P_1, P_2, \ldots, P_m \), proceeding to predictively execute \( P_1, P_2, \ldots, P_m \) and cache their results. In contrast to Apollo’s online learning capabilities, Fido requires offline training to make predictions. We provide Fido with client workload traces twice the length of the experiment interval to serve as its training set for comparison against a cold start Apollo. Additionally, we let Fido make up to 10 predictions for each matched prefix.

In all configurations, clients use session guarantees (Section 3.2) and queries executed at the remote database have their result sets immediately cached in Memcached. Thus, the difference in caching performance between the configurations is due to caching benefits provided by the query prediction engines.

Our experiments aim to answer three important questions. First, can Apollo analyze incoming queries and learn an effective prediction framework even with significant randomness within a workload? Second, are Apollo’s predictions accurate enough to avoid executions of expensive queries by satisfying them from...
predictively cached query results? Third, can Apollo’s predictive framework scale under an increasing number of clients? We present performance results in the next sections that include answers to these questions.

4.2 TPC-W Benchmark

The TPC-W Benchmark [41] generates a web commerce workload by having emulated browsers interact with servlets that serve webpages. The webpages require persistent data from storage so servlets execute database queries against the remote database to generate webpage content. The TPC-W benchmark includes 14 different web interactions for clients (e.g., Best Sellers, Order Inquiry) each with their own distinct set of queries. For a given client, the next web interaction is chosen probabilistically based on the previous interaction. We use a popular implementation [33] of the TPC-W Benchmark specification.

The TPC-W benchmark represents an important use case for Apollo, since even small changes in latency can significantly impact web traffic [20] and sales [30]. Further, it serves as a challenging workload for Apollo due to its inherent randomness and large number of different queries. This randomness serves to test the viability of Apollo’s predictive framework under a variable workload.

We generated a 33 GB TPC-W database with 1,000,000 items. We measured Apollo’s performance using the TPC-W benchmark browsing mix executed for 20 minute measurement intervals while scaling-up the number of clients using our default TPC-W parameters discussed in Section 4.7.

4.2.1 Performance Results. Figure 5(a) shows Apollo’s performance for an increasing number of clients compared to Memcached and Fido. Apollo significantly outperforms both Fido and Memcached, enjoying a large response time reduction of up to 33% over Memcached and 25% over Fido. Fido has slightly lower response time than that of Memcached due to query-instance level predictive caching, but is unable to recognize query template patterns and generalize to unseen queries, precluding it from being competitive with Apollo. In the case of Memcached, we see its warmed cache offers little advantage over Apollo’s and Fido’s cold starts — invalidation and randomness limit the effects of cache warming.

Each configuration shows a reduction in response time as the number of clients increase, a consequence of the shared cache between clients. However, shared caching is unable to compete with our predictive caching scheme as in a shared cache, a client must incur a cache miss, execute, and then store query results before others can use it. Consequently, Apollo’s techniques of query prediction and informed ADQ reload prove superior, even as the client load is scaled up.

Figure 5(b) shows the distribution of tail response times for each of the experimental configurations for 50 client TPC-W runs. Apollo’s response times are significantly lower than any of the other methods, particularly for the higher percentiles, due to an improvement in cache hits. At the 97th percentile, Apollo reduces tail latencies by 1.8x over Memcached, and 1.9x over Fido. Again, Fido tends to perform about as well as Memcached, despite its large training set size, as it cannot generalize its patterns to query templates for FDQ prediction and query reload.

Figure 5(c) shows average query response times in 4 minute intervals. We see that Apollo exhibits a downward trend in response time from the start of the measurement interval as it effectively learns query correlations and parameter mappings, resulting in an improvement of 30% over its average latency during the first four minutes. Although the other systems’ performance oscillates according to workload patterns, they do not learn query patterns — their final average query latencies are comparable to that incurred in their first few minutes.

To ensure that Apollo can provide these latency reductions without undue resource overhead, we added instrumentation to determine the time and memory needed to find and construct new FDQs. On average, it takes less than 1% of response time to discover new FDQs given a newly executed query, and less than 2% of response time to construct an FDQ. We have observed that Apollo uses scant system resources, requiring only 1.5% the amount of memory used by the database for tracking the transition graph and query parameter mappings. With these meager resources, Apollo predictively executes and keeps important queries in cache, resulting in lower average query response times than both Fido and Memcached.

To answer the performance questions we had posed earlier in Section 4.1, Apollo is indeed able to make accurate and fruitful predictions for what to cache, predicting and retaining important result sets in the cache for longer in the face of randomness in the TPC-W benchmark workload without significant computation or memory overhead.

4.3 TPC-C Benchmark

The TPC-C Benchmark emulates an order-entry environment in which multiple clients execute a mix of transactions against a database system [1]. Each of these clients functions as a store-front terminal, which submits orders for customers, confirms payments, and updates stock levels. In contrast to TPC-W’s workload, TPC-C’s OLTP workload features many short-running queries which avoid contention by reduced locking of significant parts of the
The TPC-C specification has two read-only transactions, Stock Level and Order Status, both of which present opportunities for predictive execution. Since the goal of our experiment with TPC-C is to show the scalability of predictive execution under high numbers of clients, we scale up the mix of read-only transactions to 95% with updates making up the remaining 5%. In doing so, Apollo must track, construct, and execute far more predictive queries than in the TPC-W experiments. Thus, this experiment’s purpose is to show how well Apollo can handle hundreds of clients executing predictive queries simultaneously.

In our experiments, we use the OLTPBench TPC-C implementation from Difallah et al. [17]. To properly assess scalability, we modified the read/write mix, with a final percentage of 5% Payments, 47.5% Order Status, and 47.5% Stock Level Transactions. This mix forces the prediction engines to construct and execute significantly more predictive queries.

A TPC-C database of size 100 GB with 1000 warehouses was generated and loaded into a US-West MySQL instance using the data generation mechanism of OLTPBench. For the following experiments, we used our default TPC-C parameters (discussed in Section 4.7) with a 5% write mix. We choose the warehouse parameter in our queries according to a uniform distribution, which results in more predictive executions than a skewed Zipf distribution — recall that Apollo will not predictively execute queries that are already cached (Section 3.3).

4.3.1 Performance Results. Figure 6 shows the scalability of Apollo, Fido, and Memcached for increasing numbers of clients. Apollo exhibits a significantly lower average response time than Memcached and Fido, even as the number of clients, and therefore the number of predictive query executions, increases. Apollo’s efficient structures and algorithms for tracking and prediction allow scaling even with a large number of clients. Fido and Memcached perform about the same, even as we increase the number of clients. With a large database, query parameters are highly variable and rarely repeated, causing Fido’s non-template approach to see few queries from its training set and in turn reducing prediction accuracy. As the number of clients increase, the positive effects of shared caching dwarf that of Fido’s predictions, resulting in similar performance characteristics between Fido and Memcached.

These results show that Apollo can deliver significant performance gains while scaling effectively to hundreds of concurrent clients continuously executing predictive queries.

Figure 6: Experiment results for 20 minute TPC-C runs using Apollo, Fido, and no prediction engine.

Figure 7: Experiment results for changing the workload from TPC-C to TPC-W using Apollo, Fido, and no prediction engine.

4.4 Adapting to Changing Workloads

To assess Apollo’s ability to adapt to changing workloads, we conducted an experiment in which the workload was changed from our TPC-C workload described in Section 4.3 to TPC-W partway through the experiment (Figure 7). We see that Apollo quickly learns predictions for the TPC-C workload, resulting in the performance gains shown in Figure 6. By contrast, Fido and Memcached have relatively constant performance during the TPC-C run since they are unable to generalize and make effective predictions for upcoming queries (Section 4.3).

Once the workload switches, shown by a vertical dashed line in Figure 7, each configuration experiences a brief penalty in performance since the predictive engines cannot make any predictions for queries in the new workload, and no TPC-W queries are cached. However, Apollo quickly returns to its typical performance on TPC-W (Figure 5(a)) since it uses online learning to discover query patterns. Fido and Memcached perform similarly after the switch since Fido is unable to make predictions for an untrained workload. Although Fido was trained for TPC-C in this experiment, we note that its performance is comparable to an appropriately trained Fido on the TPC-W portion. This observation further highlights the ineffectiveness of Fido’s prediction scheme for the correlated query patterns, which Apollo excels at predicting.

4.5 Geographic Latency Testing

To assess the effects of different geographic latencies between Apollo and the database, we deployed TPC-W databases in the US-East and Canada regions. Because Apollo, the cache and the benchmark machine are all located in the US-East region, the first configuration tests a “local” deployment, in which latency among the machines is minimal (a few milliseconds). The second configuration tests moderate latencies of 20 ms.

In both configurations (Figures 8(a) and 8(b)), we see that Apollo preserves its lead over the other systems despite limited geographic latency. Apollo reduces query latency by up to 50% in the US East region and by up to 40% in the Canada region. This improvement in the performance gap compared to the higher latency experiments is because cache misses in low latency environments have a larger effect on average performance than when latency is high. The reason for this effect is that Apollo’s advantage when caching expensive queries becomes even more significant with reduced latency; prioritizing expensive and frequently executed queries results in a substantial improvement and failure to predictively cache them (as in LRU and Fido’s case) results in a relatively larger performance degradation.
These results are not to be interpreted as Apollo is “best” in a local setting with near zero latency — the total latency savings for the remote settings are larger than that of the local setting. Apollo provides substantial reductions in average and total latencies in both settings, resulting in enhanced user experience.

4.6 Multiple Apollo Instances

Apollo can scale to high loads by partitioning clients among multiple Apollo engine instances and cache stores. Each Apollo engine maintains a consistent session for each client connected to it, without interacting with the other instances or a centralized manager. As such, Apollo can scale indefinitely, subject to the underlying database’s capacity.4

To determine Apollo’s scaling characteristics, we deployed Apollo on less powerful m4.xlarge EC2 instances with 4 vCPUs and 16 GB of RAM. We use these less powerful machines as they are individually unable to handle large numbers of clients, necessitating a horizontal scale-out for Apollo instances. We test three different Apollo configurations: one with a single Apollo instance, another with two Apollo instances, and a third with three Apollo instances. Each Apollo instance is given its own dedicated cache, thereby avoiding the need to synchronize version vectors across instances to preserve client sessions. Clients are evenly distributed and pinned to Apollo instances.

The results of the experiment are shown in Figure 8(c). As the client load increases, we see that it quickly overwhelms the 1-instance Apollo configuration, resulting in a large increase in query latency. The 2-instance and 3-instance Apollo configurations show significantly improved scalability, though eventually the 2-instance configuration begins to show a similar upward trend in latency.

We observed that the two instance query latency at 20 clients is slightly lower than that of the 3-instance configuration. This effect is primarily due to splitting the clients across three machines rather than two, resulting in the 3-instance configuration receiving less queries to learn from. ADQs and FDQs are shared among clients, which results in longer learning times to reach a steady state with fewer clients. With a larger number of clients, the increase in the amount of data to learn from and the computing power available result in improved performance over the 2-instance configuration.

Although having multiple Apollo instances share models and training data would reduce the effects of training data splitting, the trade-off is increased synchronization between otherwise independent nodes. We eschew this approach for two reasons. First, clients should only be split across Apollo instances when a single instance cannot handle the load. As seen in Figure 8(c), Apollo receives enough training data from clients well before it reaches its load capabilities, even on a less powerful machine. Second, slightly increased training times are a small price to pay for horizontal scalability. Addressing scalability issues in production systems is challenging — learning for a few more minutes is a simple and inexpensive solution to the insufficient data problem.

These multi-instance experiments demonstrate Apollo’s ability to scale to large numbers of clients through horizontal scaling and client-session consistency semantics. Since separate Apollo instances do not need to communicate, Apollo demonstrates excellent scaling characteristics.

4.7 Sensitivity Analysis

A key feature of Apollo is its ability to be configured to operate under different workloads and system deployments. This configurability is enabled by the provision of parameters that can be set according to a particular workload and deployment. In this section, we discuss the effects of these parameters on the overall performance of the system as well as our choice of their default settings.

For TPC-W, our default parameter choices were: \( \Delta t = 15s \), \( \tau = 0.01 \), and a reload cost threshold of \( \alpha = 0 \). As per the specification [41], TPC-W uses randomized think times with a mean of 7s. Each client has their own application state, which is determined through a probabilistic transition matrix. Therefore, a client’s web interactions do not generate a pre-determined chain of queries.

The maximal time separation \( \Delta t \) and minimum probability threshold \( \tau \) are correlated. As \( \Delta t \) decreases, the probability of correlated query templates executing within this interval also decreases, thereby requiring a lower \( \tau \) value to capture the relationship between query templates. Similarly, as \( \Delta t \) increases, the probability of two correlated query templates executing within this interval also increases, so higher \( \tau \) values are sufficient. If both \( \Delta t \) and \( \tau \) are small then it is likely that spurious relationships between query templates will be discovered, but these spurious relationships are filtered out by Apollo’s parameter mapping verification period, and are therefore seldom predictively executed. Since TPC-W’s workload bottlenecks on the database, Apollo filters out the spurious correlations in exchange for discovering as many relationships as possible. To do so, we set a high value of \( \Delta t = 15s \) and a low threshold of \( \tau = 0.01 \). These values were empirically confirmed to yield the best results.

---

4 Standard database scaling techniques (partitioning/replication) can be applied to increase database capacity and performance.
In Section 3.4.2, we defined $\alpha$ to be the minimum cost that an ADQ must have to be reloaded. Note that the cost of an ADQ is the average response time multiplied by the probability of the query executing. Hence as $\alpha$ is increased, only ADQs that are both popular and expensive are reloaded. We experimented with different values of $\alpha$ and found that for small values of $\alpha$ (less than 5% of the average query response time), there was little change in query response time. However, as $\alpha$ continued to be increased past this threshold, the average query response grew by over 10%, as valuable ADQs were not reloaded. To ensure that all ADQs were reloaded, $\alpha$ was set to 0 in our experiments.

Our default parameter choices for TPC-C were the same as our choices for TPC-W, with the exception of $\Delta t = 7s$. We observed similar trends with $\Delta t$ and $r$ in TPC-C as in TPC-W, and simply adjusted the scale of $\Delta t$ to match the workload 3s think time. These values were empirically confirmed to yield the best results.

In our experiments, we used a cache 5% the size of the database. We observed that increasing the cache beyond this size did not affect the relative performance differences between Apollo, Memcached, and Fido.

5 RELATED WORK

The closest related work to Apollo is that of Fido [32], which uses associative memory for predictive prefetching in a traditional client/server database system. In addition to the details about Fido in Section 4.1, query patterns in Apollo are tracked at the query template level so a single relationship in Apollo can map to many in Fido. Tracking individual data object accesses, or parameterized queries, means that if Fido has not previously seen a particular parameterized query it will not be able to make a prediction. In contrast, if Apollo has seen the query template (regardless of parameters), it can infer correlation between queries and predictively execute. By condensing multiple stored patterns into a single query template relationship, Apollo is also able to conserve space and in turn store and execute a larger variety of predictive queries. As Fido requires offline training, it cannot adapt to changes in object access patterns. As we have shown, the online nature of Apollo’s Markov model allows it to dynamically change over time and thus adapt to new query patterns.

Keller et al. [24] describe a distributed caching system for databases, which uses approximate cache descriptions to distribute update notifications to relevant sites and execute queries locally on caches. Each site’s cached data is tracked using query predicates. Apollo differs from this work in that we focus on the predictive execution of consequent queries derived from query patterns, which Keller et al. do not consider.

Scalpel [9] can track patterns in queries at only a database cursor level, intercepting open, fetch and close cursor operations within the JDBC protocol. Since Scalpel must intercept JDBC API calls and the JDBC API is translated to database specific protocols, Scalpel can be deployed as a cache only at the client — a single Scalpel instance cannot handle clients from multiple physical machines nor can it share the cache and learn from multiple client instances. Scalpel must also be trained offline to find patterns, re-write and prefetch query results, and its cached results are invalidated on writes and at the start of new transactions. In contrast, Apollo manages a consistent cache shared across clients and transactional boundaries. Scalpel is restricted to client-side deployment and does not support shared caching for multiple clients, making it uncomparable to Apollo.

Pavlo et al. [6] implement Markov models in H-Store and use them to optimize transaction execution for distributed database physical design. The system constructs a series of Markov models for stored procedures and monitors the execution paths under a set of input parameters. Their model can be leveraged for optimizations such as determining which partition a procedure will access most frequently, selecting it as the base partition for execution. Further, they use these models to predict the partitions a distributed query will need to access and only lock those partitions. The overhead associated with locking all possibly accessed partitions and that of a fine-grained locking scheme are avoided. Apollo operates beyond this stored procedure context, and provides benefits through caching future queries, not by analyzing query execution paths. Moreover, Apollo learns query correlations for query execution and not transaction execution for distributed partitions.

DBProxy [5] is a caching system developed by IBM to cache query results on edge nodes. DBProxy uses multi-layered indexes and query containment to match queries to results, evicting stale and unused results. Its single session guarantees differ from Apollo’s per-client sessions and limit scalability, in addition to not using online learning or predictive caching to improve performance.

Ramachandra et al. [34] propose a method for semantic prefetching by analyzing the control flow and call graph of program binary files. Given the source code for a database application, the system analyzes and modifies it, adding prefetch requests into the code as soon as the parameters are known and query execution guaranteed. Since this work is limited to requiring access to the source code of application binaries, it only works for fixed workloads. Because Apollo analyzes query streams, it is able to adapt to changing query patterns over time.

Although proprietary middleware caching solutions have been developed [8, 15, 26], they do not use predictive analytics to identify future queries and preload them in the cache.

Scheuermann et al. [38] propose the Watchman cache management system, which uses query characteristics to improve cache admission and replacement algorithms. Unlike Apollo, Watchman does not discover query patterns for use in predictive execution and instead focuses solely on cache management.

Holze et al. [23] have broached the idea of modeling workloads using Markov models, but such work focuses only on determining when an application’s workload has been altered rather than relying on statistical models for caching purposes. They suggest a Markov model as a means to achieve an autonomic database, enabling features such as self-configuration, self-optimization, self-healing, and self-protection. In contrast, Apollo uses Markov models of user workloads to predict future queries and enables predictive query caching.

Promise [35] is a theoretical framework for predicting query behavior within an OLAP database. Promise uses Markov models to predict user query behaviour by developing state machines for parameter value changes and transitions between OLAP queries, but does not consider direct parameter mappings, FDQ hierarchies, or pipelining predictions. Unlike Apollo, Promise does not demonstrate validation of its techniques through a system implementation.

In the view selection problem [11], one must decide on a set of views to materialize so that execution of the workload minimizes some cost function and uses fixed amount of space. Most work in this area requires knowledge of the workload a priori [2, 22],
with the remainder not considering machine learning techniques for uncovering patterns for use in view selection [16, 25].

XML XPath templates have some similarities to query templates [31], but are not used for online learning in predictive execution and caching. Instead, XPath views are selected using offline training in a warm-up period [29, 42], similar to that of Fido [32]. Similar ideas have been explored to cache dynamic HTML fragments [13].

6 CONCLUSION

In this paper, we propose a novel method to determine and leverage hidden relationships within a database workload via a predictive learning model. We present the Apollo system, which exploits query patterns to predictively execute and cache query results. Apollo’s online learning method makes it suitable for different workloads and deployments. Experimental evaluation demonstrates that Apollo is a scalable solution that efficiently uses a cache and outperforms both Memcached, an industrial caching solution for different workloads and the popular Fido predictive cache.

REFERENCES


