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

Brad Glasbergen
University of Waterloo
bjglasbe@uwaterloo.ca

Michael Abebe
University of Waterloo
mtabebe@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
kdaudjee@uwaterloo.ca

Scott Foggo
University of Waterloo
sjfoggo@uwaterloo.ca

Anil Pacaci
University of Waterloo
apacaci@uwaterloo.ca

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 to workload changes and its 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 [11, 13, 41]. 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 [38, 39] 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 close to clients. 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 datacenters can be quite large. Efficiently caching data on these edge nodes substantially reduces request latency for clients.

Existing caching solutions for edge nodes and content delivery networks (CDN) focus largely on static data, necessitating costly round trips to remote data centers for requests relying on dynamic data [21]. Since a majority of webpages today are generated dynamically [5], a large number of requests are not satisfied by cached data, thereby incurring significant 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.

(a) Datacenter Locations   (b) Edge Node Locations
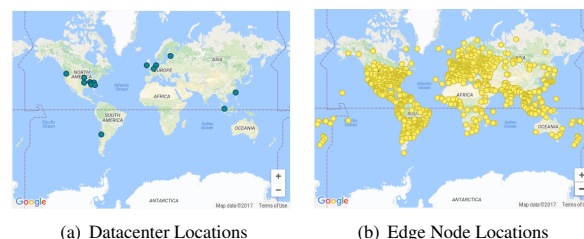
Figure 1: Google's datacenter and edge node locations [21].

```
1. SELECT  C_ID   FROM CUSTOMER WHERE
C_UNAME = @C_UN and C_PASSWD = @C_PAS

2. SELECT  MAX(O_ID)   FROM ORDERS WHERE
O_C_ID =  @C_ID

3. SELECT ... FROM ORDER_LINE, ITEM
WHERE OL_I_ID = I_ID and OL_O_ID =  @O_ID
```

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

Database client workloads often exhibit query patterns, corresponding to application usage patterns. In many workloads [1, 10, 42], 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 benchmark [42]. 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
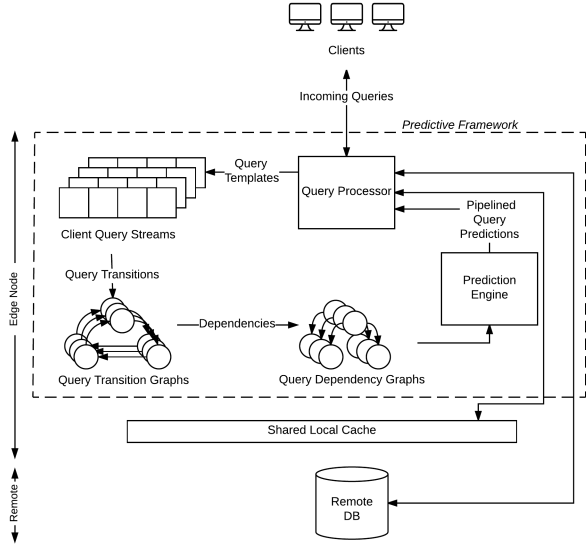
Figure 3: Query flow through components of the predictive framework.

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 applications, have been previously described [10].

In this paper, we propose a novel prediction framework that uses a query-pattern aware technique to improve performance in geo-distributed database systems through caching. We implement this framework in Apollo, which uses a query transition graph to learn correlations between queries and to predict future queries. In doing so, Apollo determines query results that should be cached ahead of time so that future queries can be satisfied from a cache deployed close to clients. Apollo prioritizes common and expensive queries for caching, eliminating or reducing costly round-trips to remote data without requiring modifications to the underlying database architecture. Apollo's ability to learn allows it to rapidly adapt to workloads in an online fashion. Apollo is designed to enhance an existing caching layer, providing predictive caching capabilities for improved performance.

The contributions of this paper are threefold:

(1) We propose a novel *predictive* framework to identify relationships among queries and predict consequent ones. Our framework uses *online learning* to adapt to changing workloads and reduce query response times (Section 2).

(2) We design and implement our framework in a system called Apollo, which predictively executes and caches query results on edge nodes close to the client (Section 3).

(3) We deploy and extensively test Apollo on Amazon EC2 using the TPC-W and TPC-C benchmark workloads to show that significant performance gains can be achieved for different query workloads (Section 4).

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, 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 both deployment environments benefit significantly from Apollo's predictive caching.

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 Q_1 to look up a product ID followed by query Q_2 to check the stock level of a given product ID. A common usage pattern is to execute Q_1, and then use the returned product ID as an input to Q_2 to check that product's stock level. In this case, Q_2 is directly related to Q_1 via a *dependency* relationship. Specifically, Q_2 relies on the output of Q_1 to execute.

We generalize our model by tracking relationships among query templates rather than among parameterized queries. Two queries Q_1 and Q_2 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 (Q_1, Q_1') and their corresponding templates (Qt_1, Qt_1'):

```
Q₁: SELECT C_ID FROM CUSTOMER WHERE C_UNAME
= 'Bob' and C_PASSWD = 'pwd'
Qt₁: SELECT C_ID FROM CUSTOMER WHERE C_UNAME =
? and C_PASSWD = ?
```

Q'_1: SELECT C_ID FROM CUSTOMER WHERE C_UNAME = **'Alice'** and C_PASSWD = **'pwd2'**
Qt'_1: SELECT C_ID FROM CUSTOMER WHERE C_UNAME = **?** and C_PASSWD = **?**

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, constructing it in an online fashion. We exploit the memory-less property of Markov models to simplify transition probability computations — transition probabilities are based solely on the previous query the client executed.

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

The τ parameter serves as a configurable confidence threshold for query template relationships. More concretely, the τ parameter provides the minimum required probability for Qt_j executing after Qt_i to infer that they are related. By choosing τ appropriately, we can limit the predictive queries executed after seeing Qt_i 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.

$P(Qt_j|Qt_i)$ is too broad to capture 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. 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, Δt, which specifies the maximum allowable time separation between related query templates.

Definition 2.1. For any two query templates Qt_i, Qt_j, in which Qt_j is executed T time units apart from Qt_i, if $P(Qt_j|Qt_i; T \leq \Delta t) > \tau$ for some threshold parameter $\tau \in [0, 1]$, we consider Qt_j to be a related query template of Qt_i.

To learn a transition graph representing $P(Qt_j|Qt_i; T \leq \Delta t)$, we map executed queries to query templates and place them at the tail 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 Δt of the first query template, adding counts to their corresponding edges and afterwards incrementing the vertex count indicating number of times the template has been seen. To calculate the probability of $P(Qt_j|Qt_i; T \leq \Delta t)$, we take the edge count from Qt_i to Qt_j and divide by the vertex count for Qt_i. To use

---

Algorithm 1 Query Transition Graph Construction

Input: $(Qt_1, t_1), (Qt_2, t_2), \ldots$, an infinite stream of incoming query template identifiers and their execution timestamps,
Δt, a fixed time duration,
$G = (V, E)$, a directed graph, initially empty,
$w_v : V \rightarrow \mathbb{N}$, vertex counters indicating the number of times we have seen the vertex, initially all zero,
$w_e : V \times V \rightarrow \mathbb{N}$, edge counters indicating the number of times we've seen the outgoing vertex followed by the incoming vertex within Δt, initially all zero.

```
i ← 1
loop
    if t_i + Δt >now() then
        wait until now()> t_i + Δt
    end if
    V ← V ∪ {Qt_i}
    w_v(Qt_i) ← w_v(Qt_i) + 1
    j ← i + 1
    loop
        if t_j > t_i + Δt then
            // too far apart in time
            break
        else
            E ← E ∪ {(Qt_i, Qt_j)}
            w_e(Qt_i, Qt_j) ← w_e(Qt_i, Qt_j) + 1
        end if
        j ← j + 1
    end loop
    // advance forward in stream
    i ← i + 1
end loop
```

the variables directly from Algorithm 1, the probability that query template Qt_j executes within Δt of a query template Qt_i is given by $\frac{w_e(Qt_i, Qt_j)}{w_v(Qt_i)}$. Per Definition 2.1, if this probability exceeds τ then Qt_j is considered related to Qt_i.

The choice of the Δt parameter can impact prediction efficacy. If Δt is too high, it is possible that relationships will be predicted where there are none; if Δt is too low, we may not discover relationships where they are present. Although the choice of Δt is workload dependent, some indicators aid us in choosing an appropriate value, such as query arrival rate. If $P(Qt_j|Qt_i; T \leq \Delta t)$ is high for a fixed Qt_i and many different Qt_j, then either Qt_i is a common query template with many quick-executing related query templates, or Δt is set too high. If this holds for many different Qt_i, then Δt can be decreased. A similar argument holds for increasing Δt. We discuss selection of Δt and τ 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 adapts to learn the changed workload. Moreover, online learning precludes the need to undergo expensive offline training before deployment. Instead, our model rapidly learns client 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 Qt_1 and the template for the second query as Qt_2. In the first stage of tracking, we observe which query templates have executed within Δt of Qt_1. Once Qt_1 has executed enough times (according to the verification period), we begin to construct mappings among the query templates. After Qt_1 finishes an execution, we record its output set. When any of Qt_1's related query templates (in this case assume only Qt_2) are executed, we record their input sets. We then check if any column's result in Qt_1's output set maps to the input parameters of Qt_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.[1] 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 Qt_2 and Qt_3 and use them to execute Qt_3 given Qt_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. 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.
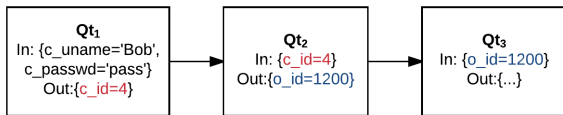

Figure 4: An example of pipelining within a dependency hierarchy. The arrows represent a mapping from a prior query template's output set to the consequent query template's input set.

Figure 4 illustrates how pipelining can be used to form extended chains of predictive executions using the TPC-W example from Figure 2. Qt_1 has a mapping to Qt_2, which in turn has a mapping

to Qt_3. If Qt_1 is executed, we can forward its result set as input with which to predictively execute Qt_2. Once Qt_2 has also been executed, we can predictively execute Qt_3. As such, Qt_2 is *fully defined* given the result set of Qt_1, and Qt_3 is fully defined given the result set of Qt_2. We formalize the notion of fully defined queries:

Definition 2.2. A fully defined query template (FDQ) Qt_j has all of its inputs provided by some set, possibly empty, of prior query templates $Qt_{i_1}, Qt_{i_2}, \ldots, Qt_{i_k}$ where each Qt_{i_m} ($\forall m \in [1, k]$) is either:

(1) a fully defined query template, or
(2) a dependency query template, required to execute Qt_j.

Per Definition 2.2, both Qt_2 and Qt_3 are FDQs, but Qt_1 is simply a dependency query. This definition captures 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 Qt
  record_query_template(Qt)
  new_fdqs = find_new_fdqs(Qt)
  rdy_fdqs = mark_ready_dependency(Qt)
  rdy_fdqs = rdy_fdqs ∪ new_fdqs
  ordered_fdqs = find_all_runnable_fdqs(rdy_fdqs)
  for all rdy_fdq ∈ ordered_fdqs **do**
    execute_fdq(rdy_fdq)
  end for

Discovering new FDQs, managing FDQ dependencies, and pipelining predictions comprise the main routine of the predictive framework. The engine executes Algorithm 2 after the execution of a client-provided instance of query template Qt. The engine records Qt's result set and input parameters in the query transition graph (Section 2.3), looks for parameter mappings, and records discovered dependencies in the dependency graph. This query template is then marked as executed so that FDQ pipelines that depend on its result set can proceed. Any queries that are determined ready for execution given the result of this query (and previously executed queries) are then executed, forwarding parameters from their dependent queries' result sets. 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 (recursively) 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

---

[1] If 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, our 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. Each executed read-only 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 overall system performance. Apollo's predictive engine operates in a *complementary* manner where queries are passed unchanged through to the cache and database, preserving the effective workload behaviour. Apollo executes predicted queries and caches them ahead of time, 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 [33] database. The JDBC API [32] makes Apollo database agnostic and therefore portable, allowing MySQL to be easily swapped for any other JDBC compliant relational database system.

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 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 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 Qt_i) since these are the templates that may have new mappings from Qt_i's result set. Qt_i itself is also

checked since it may be an ADQ (if it has no input parameters). For each query template Qt_j 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 Qt_j'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 Qt_j and its dependencies are recorded in the dependency graph. For efficiency, we represent the dependency graph as a hash map 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 Qt_i
Output: a set of newly discovered FDQs
  queries_to_check = get_related_queries(Qt_i)
  queries_to_check = queries_to_check ∪ {Qt_i}
  new_fdqs = {}
  for all Qt_j ∈ queries_to_check **do**
    if !already_seen_deps(Qt_j) **then**
      p_mappings = get_prior_query_mappings(Qt_j)
      if have_enough_mappings(Qt_j) **then**
        fdq = construct_fdq(Qt_j,p_mappings)
        unresolved_deps = get_dependencies(fdq)
        add_to_dep_graph(unresolved_deps, fdq)
        mark_seen_deps(fdq)
        new_fdqs = new_fdqs ∪ {fdq}
      end if
    end if
  end for
  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 that contain it. To do so, we track the FDQs that the system has constructed before, returning a 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. After the execution of a given query template Qt_i, each dependent FDQ marks that dependency as satisfied. If all of an FDQ's 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. Apollo determines which FDQs are executable given the current system state and a newly executed query, adding them to the list of ready FDQs. Apollo then determines which other FDQs are executable given this FDQ list, repeating the process as necessary. This final list of FDQs is then executed in order, feeding result sets as parameters to dependent FDQs.

Algorithm 4 mark_ready_dependency

Input: an executed query Qt_i whose result set is now available
Output: a set ready_fdqs of FDQs ready for execution
  ready_fdqs = { }
  dependency_lists = get_dep_query_dlists(Qt_i)
  for all d_list ∈ dependency_lists **do**
    mark_dependency_satisfied(d_list, Qt_i)
    if all_deps_satisfied(d_list) **then**
      ready_fdqs=ready_fdqs ∪ get_fdq(d_list)
      reset_dependencies(d_list)
    end if
  end for
  return ready_fdqs

3.2 Client Sessions

Apollo uses a client session consistency scheme [15], enabling its predictive cache to share cached results among clients and scale in the presence of write queries. In brief, each client has an independent session that guarantees that it accesses data at least as fresh as data it last read or wrote and that it efficiently shares cached entries with other clients.

Each client maintains a version vector $(v_1, v_2, \ldots, v_n)$ indicating its most recently accessed version v_i for each table R_i. Query results are stored in the cache and timestamped with a version vector $(c_1, c_2, \ldots, c_n)$ matching the version vector of the client that wrote it. When a client wants to execute a read query on a set of tables $(R_1, R_2, \ldots, R_n)$, it checks if there exists an entry in the cache for that query with a version vector with $(c_1 \geq v_1, c_2 \geq v_2, \ldots c_n \geq v_n)$. If so, the client will retrieve and return the cached result, updating its client state for each of the tables to match that of the cached entry. If there is no such entry, the client will execute the query against the database, updating its version vector for each of the affected tables to match their versions in the database and storing the result in the cache. Write queries are never predictively executed (to prevent unnecessary rollbacks) and always execute against the database. After a client executes a write query, its version vector is updated to match the state of the database.

Since cache misses and write queries update a client's version vector, old cache entries may be stale under the client's new version vector. Therefore, if it is important to update a client's version vector only when strictly necessary, and by the minimum amount. As such, when a client could read two different versions of a cached key, Apollo will return the value for the cached key with a version vector that minimizes the distance from the client's version vector. Apollo uses a variety of optimizations to reduce the impact of write queries on predictive caching and system performance, discussed in Section 3.4.

Since a client's session is independent of the sessions of other clients, Apollo can easily scale horizontally. An individual client must route all of its requests to the same Apollo instance to maintain its session, but other clients and processes do not affect its session guarantees. Thus, extract, transform, load (ETL) processes, database triggers, and client write requests do not result in mass invalidations of cached data. Furthermore, Apollo instances do not need to communicate with each other to maintain sessions because a client's session is tracked by a single Apollo instance.

3.3 Publish–Subscribe Model

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 read query, Apollo consults a hash map to determine if a copy of the query is already executing. If so, Apollo blocks the query until the other query returns, passing along its result set. Otherwise, it will record an entry in the hash map 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 any time, including shared predictive query pipelines for multiple clients.

When Apollo determines that a client's query has multiple usable versions of its results cached, Apollo will 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 reading later versions will result in large version vector updates for the client and may therefore cause misses for other cached results. Similarly, if Apollo must retrieve the result set from the database, Apollo will 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 strive to 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. Recall from Section 2.2 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. Using this technique, we predict if a client will retrieve the results for a predictively executable query before its results become stale. Apollo will predictively execute and cache only query results that it deems are likely to be used.

To determine if predictively executing and caching a query's results 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 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 the time needed to execute the FDQ. In essence, this process returns the longest expected path from the child weighted by mean query runtimes. To provide an approximation of individual query runtimes, we use the mean execution time for each query

template. Although more sophisticated methods can be used [4, 45] 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 a given FDQ f has been determined (say t), Apollo looks up the client's transition graph with smallest interval Δt 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 τ threshold), we avoid executing f to save on database execution 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, 42] tend to have a small number of unique query templates, the storage overhead is minimal.

3.4.2 Informed ADQ Reload.
Write queries update a client's 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 expensive for the database to execute, we limit ADQ reload to only those predictions for query templates considered valuable according to the cost function $cost(Qt) = P(Qt) \cdot mean\_rt(Qt)$.[2] 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 α, we reload it into the cache. We discuss α 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 and storage systems, as well as the Fido predictive cache [34]. 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 [34] for query prediction, scanning client query streams to predict upcoming queries. Fido-like approaches have been employed to prefetch objects in databases [8]. Fido's implementation-independent middleware prediction engine makes 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 to 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. All results presented are the average over at least five independent runs, with bars around the means representing 95% confidence intervals.

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 the Fido prediction engine configuration [34]. 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 also load query results into the cache after they execute a read-only query on the remote database, but Apollo uses the predictive framework from Section 2 and Fido uses its own predictive engine, which is detailed below.

Unlike Apollo, Fido functions on an individual query level rather than on 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 $Q_1, Q_2, \ldots, Q_n$, $P_1, P_2, \ldots, P_m$, 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 key questions. First, can Apollo analyze incoming queries and learn patterns within a workload? Second, are Apollo's predictive caching capabilities effective in reducing query round-trip time by avoiding costly database query executions? Third, can Apollo's predictive framework scale with 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 [42] 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
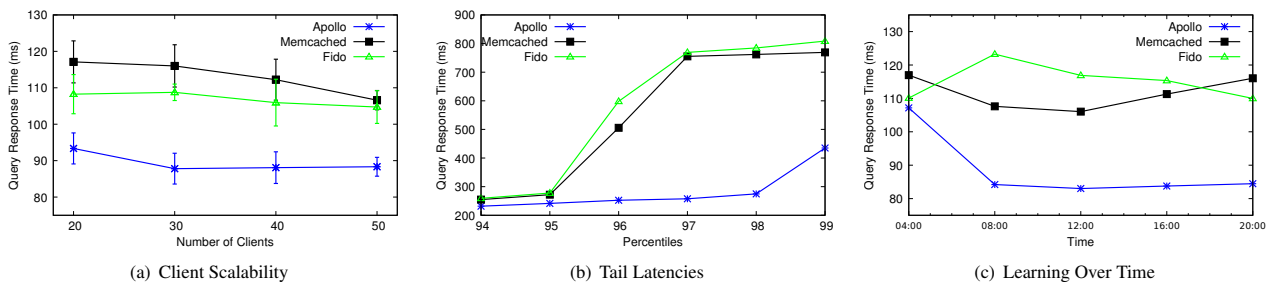
---

[2]Note that the techniques in Section 3.3 apply; shared query dependencies and overlapping client query submissions will not result in multiple executions of ADQs.

| (a) Client Scalability | (b) Tail Latencies | (c) Learning Over Time |

Figure 5: Experiment results for 20 minute TPC-W runs using Apollo, Fido, and Memcached (no prediction engine).

client, the next web interaction is chosen probabilistically based on the previous interaction. We use a popular implementation [35] 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 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 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 response time 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 response times are comparable to that incurred in their first few minutes.

To ensure that Apollo can provide these response time 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. Apollo's predictive techniques submit an additional 25% more queries to the remote database compared to the Memcached configuration. Apollo's intelligent query caching techniques place little additional load on the remote database and use meager resources, while still providing substantially 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 useful predictions for what to cache, predicting and retaining important result sets in the cache for longer 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 database. As such, the TPC-C benchmark serves to directly test the scalability of Apollo.

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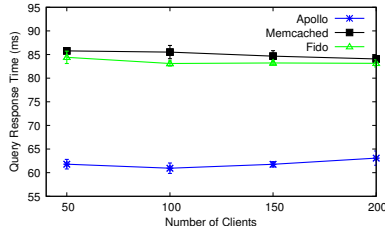
Figure 6: Experiment results for 20 minute TPC-C runs using Apollo, Fido, and Memcached (no prediction engine).
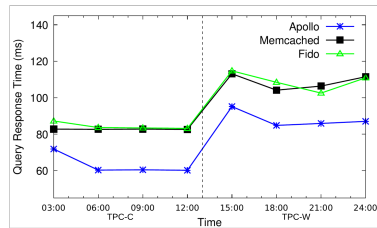

Figure 7: Experiment results for changing the workload from TPC-C to TPC-W using Apollo, Fido, and Memcached (no prediction engine).

In our experiments, we use the OLTPBench TPC-C implementation from Difallah et al [18]. 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 data 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.

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 dashed vertical line in Figure 7, each configuration experiences a brief penalty in performance because 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 response time 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 Memcached's 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* response time savings for the remote settings are larger than that of the local setting. Apollo provides substantial reductions in average and total response time in both settings, resulting in an 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 session manager.

To determine Apollo's scaling characteristics, we deployed Apollo on less powerful m4.xlarge EC2 instances with 4 vCPUs
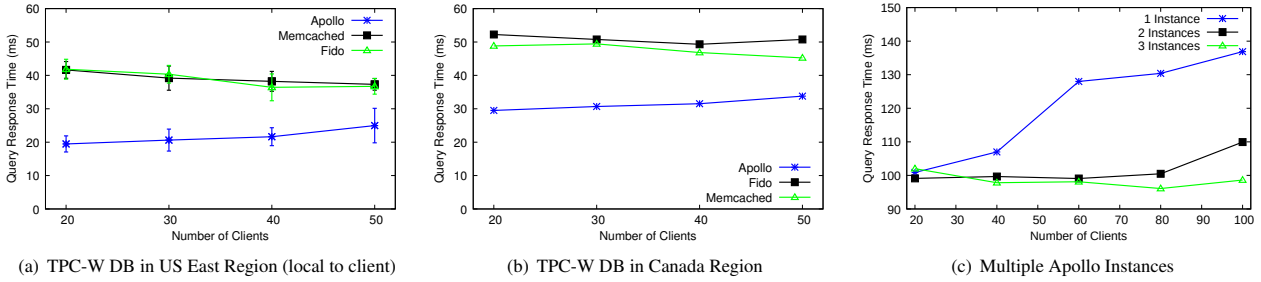
Figure 8: Experiment results for 20 minute TPC-W runs in different geographic regions and when using multiple Apollo instances.

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 maintain 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 response time. 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 response time.

We observed that the 2-instance query response time 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 fewer queries to learn from. ADQs 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 be split across Apollo instances only 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 $\alpha = 0$. Per the specification [42], TPC-W uses randomized think times with a mean of 7s. Each client has its 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 Δt and minimum probability threshold τ are correlated. As Δt decreases, the probability of correlated query templates executing within this interval also decreases, thereby requiring a lower τ value to capture the relationship between query templates. Similarly, as Δt increases, the probability of two correlated query templates executing within this interval also increases, so higher τ values are sufficient. If Δt is large and τ is small then it is likely that spurious relationships between query templates will be discovered, but such 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.

In Section 3.4.2, we defined α to be the minimum cost that an ADQ must have to be reloaded. Note that the cost of an ADQ is the mean response time multiplied by the probability of the query executing. Hence as α is increased, only ADQs that are both popular and expensive are reloaded. We experimented with different values of α and found that for small values of α (less than 5% of the mean query response time), there was little change in query response time. However, as α continued to be increased past this threshold, the mean query response time grew by over 10%, as valuable ADQs were not reloaded. To ensure that all ADQs were reloaded, α was set to 0 in our experiments.

We observed similar trends with Δt and τ in TPC-C as in TPC-W; therefore, our default parameter choices for TPC-C were the same. We left Δt large and τ small to place additional pressure on Apollo's parameter mapping filtering functionality. 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

Fido [34], detailed in Section 4.1, uses an associative memory model for predictive prefetching in a traditional client/server database system. 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. 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 [10] tracks queries at a database cursor level, intercepting open, fetch and close cursor operations within the JDBC protocol. Since the JDBC API is translated to database specific protocols, Scalpel functions as a client-side cache rather than a mid-tier shared cache like Apollo. Unlike Apollo's online learning model, Scalpel requires offline training to find the patterns that it uses for query rewriting and prefetching. Scalpel employs aggressive cache invalidation on writes and at the start of new transactions, which differs from Apollo's client-centric consistency model. Given that Apollo supports mid-tier shared caching across multiple clients, this makes Scalpel unsuitable for comparison against Apollo.

Pavlo et al. [7] 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 to determine a base partition for stored procedures and to lock only partitions that are predicted to be accessed during procedure execution. Apollo operates beyond this stored procedure context, and provides benefits through caching future queries rather than by analyzing query execution paths.

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. [36] 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 [9, 16, 26], they do not use predictive analytics to identify future queries and preload them in the cache.

Scheuermann et al. [40] 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, like Apollo. 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 [37] is a theoretical framework for predicting query behaviour 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 validate its techniques through a system implementation.

Recent research in approximate query processing [6, 44] has proposed using previous query results as a means for approximating the answer to future queries. These works develop statistical methods to provide accurate, approximate answers and error bounds for upcoming queries, which differs from Apollo's focus on learning parameter mappings for predictive caching.

In the view selection problem [12], 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 ahead of time [2, 22], with the remainder not considering machine learning techniques for uncovering patterns for use in view selection [17, 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, 43], similar to that of Fido [34]. Similar ideas have been explored to cache dynamic HTML fragments [14].

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

[1] February 2010. The Transaction Processing Council. TPC-C Benchmark (Revision 5.11). http://www.tpc.org/tpcc/. (February 2010).

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *PVLDB (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505. http://dl.acm.org/citation.cfm?id=645926.671701

[3] Akamai. 2010. New Study Reveals the Impact of Travel Site Performance on Consumers. https://www.akamai.com/us/en/about/news/press/2010-press/new-study-reveals-the-impact-of-travel-site-performance-on-consumers.jsp. (2010).

[4] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *2012 IEEE 28th International Conference on Data Engineering*. 390–401. https://doi.org/10.1109/ICDE.2012.64

[5] K. Amiri, Sanghyun Park, R. Tewari, and S. Padmanabhan. 2003. DBProxy: a dynamic data cache for web applications. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 821–831. https://doi.org/10.1109/ICDE.2003.1260881

[6] Christos Anagnostopoulos and Peter Triantafillou. 2017. Efficient scalable accurate regression queries in IN-DBMS analytics. *Proceedings - International Conference on Data Engineering* (2017), 559–570. https://doi.org/10.1109/ICDE.2017.111

[7] Stanley Zdonik Andrew Pavlo, Evan P.C. Jones. 2012. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *PVLDB* 5, 2 (2012), 85–96.

[8] P.A. Bernstein, S. Pal, and D.R. Shutt. 2009. Prefetching and caching persistent objects. (June 30 2009). https://www.google.com/patents/US7555488 US Patent 7,555,488.

[9] Christof Bornhövd, Mehmet Altinel, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2003. DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures. In *SIGMOD (SIGMOD '03)*. ACM, New York, NY, USA, 662–662. https://doi.org/10.1145/872757.872849

[10] Ivan Bowman and Kenneth Salem. 2004. Optimization of query streams using semantic prefetching. *SIGMOD* (2004), 179–190.

[11] Nathan Bronson, Zach Amsden, George Cabrera, et al. 2013. TAO: Facebook's distributed data store for the social graph. *Usenix ATC* (2013), 49–60. https://doi.org/10.1145/2213836.2213957

[12] Rada Chirkova, Alon Y Halevy, and Dan Suciu. 2001. A formal perspective on the view selection problem. In *VLDB*, Vol. 1. 59–68.

[13] James C Corbett, Jeffrey Dean, Michael Epstein, and Andrew Fikes. 2012. Spanner : Google ' s Globally-Distributed Database. *OSDI* (2012), 1–14. https://doi.org/10.1145/2491245

[14] Anindya Datta, Kaushik Dutta, Helen Thomas, Debra V, Krithi Ramamritham, and Dan Fishman. 2001. A comparative study of alternative middle tier caching solutions to support dynamic web content acceleration. In *In International Conference on Very Large Data Bases (VLDB*. 25.

[15] Khuzaima Daudjee and Kenneth Salem. 2004. Lazy Database Replication with Ordering Guarantees. In *ICDE*. 424–435.

[16] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. 2000. A Middleware System Which Intelligently Caches Query Results. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '00)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 24–44. http://dl.acm.org/citation.cfm?id=338283.338285

[17] Prasad M Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F Naughton. 1998. Caching multidimensional queries using chunks. In *ACM SIGMOD Record*, Vol. 27. ACM, 259–270.

[18] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An extensible testbed for benchmarking relational databases. *PVLDB* 7, 4 (2013), 277–288.

[19] Brad Fitzpatrick. 2016. MemCached. (4 2016). https://memcached.org/ Memcached Caching Software.

[20] Brady Forest. 2009. Bing and Google Agree - Slow Pages Lose Users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html. (2009).

[21] Google. 2017. Google's Edge Network. https://peering.google.com/infrastructure. (2017).

[22] Himanshu Gupta. 1997. Selection of Views to Materialize in a Data Warehouse. In *ICDT (ICDT '97)*. Springer-Verlag, London, UK, UK, 98–112. http://dl.acm.org/citation.cfm?id=645502.656089

[23] Marc Holze and Norbert Ritter. 2007. Towards Workload Shift Detection and Prediction for Autonomic Databases. In *Proceedings of the ACM First Ph.D. Workshop in CIKM (PIKM '07)*. ACM, New York, NY, USA, 109–116. https://doi.org/10.1145/1316874.1316892

[24] Arthur M. Keller and Julie Basu. 1996. A Predicate-based Caching Scheme for Client-server Database Architectures. *VLDBJ* 5, 1 (Jan. 1996), 035–047. https://doi.org/10.1007/s007780050014

[25] Yannis Kotidis and Nick Roussopoulos. 1999. DynaMat: a dynamic view management system for data warehouses. In *ACM SIGMOD Record*, Vol. 28. ACM, 371–382.

[26] Per-Ake Larson, Jonathan Goldstein, and Jingren Zhou. 2004. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *ICDE (ICDE '04)*. IEEE Computer Society, Washington, DC, USA, 177–.

[27] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: Comparing Public Cloud Providers. *Proceedings of the 10th annual conference on Internet measurement - IMC '10* (2010), 1. https://doi.org/10.1145/1879141.1879143

[28] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2670979.2670988

[29] Kostas Lillis and Evaggelia Pitoura. 2008. Cooperative XPath Caching. In *SIGMOD (SIGMOD '08)*. ACM, New York, NY, USA, 327–338. https://doi.org/10.1145/1376616.1376652

[30] Greg Linden. 2006. Make Data Useful. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf. (2006).

[31] Bhushan Mandhani and Dan Suciu. 2005. Query Caching and View Selection for XML Databases. In *PVLDB (VLDB '05)*. VLDB Endowment, 469–480. http://dl.acm.org/citation.cfm?id=1083592.1083648

[32] Oracle. 2017. Java SE 8 JDBC API. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/. (2017).

[33] Oracle. 2017. MySQL. https://www.mysql.com/. (2017).

[34] Mark Palmer and Stanley B. Zdonik. 1991. Fido: A Cache That Learns to Fetch. In *VLDB (VLDB '91)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 255–264.

[35] Jose Pereira. 2016. TPC-W Implementation. (4 2016). University of Minho's implementation of TPC-W.

[36] Karthik Ramachandra and S. Sudarshan. 2012. Holistic Optimization by Prefetching Query Results. In *SIGMOD (SIGMOD '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2213836.2213852

[37] Carsten Sapia. 2000. PROMISE: Predicting query behavior to enable predictive caching strategies for OLAP systems. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 224–233.

[38] Mehadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. 2009. The Case for VM-Base Cloudlets in Mobile Computing. *Pervasive Computing* 8, 4 (2009), 14–23. https://doi.org/10.1109/MPRV.2009.82

[39] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. 2014. Cloudlets: at the Leading Edge of Mobile-Cloud Convergence. *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services* (2014), 1–9. https://doi.org/10.4108/icst.mobicase.2014.257757

[40] Peter Scheuermann, Junho Shim, and Radek Vingralek. 1996. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 51–62. http://dl.acm.org/citation.cfm?id=645922.758367

[41] Jeff Shute, Mircea Oancea, Stephan Ellner, et al. 2012. F1: the fault-tolerant distributed RDBMS supporting Google's ad business. In *SIGMOD*. 777–778.

[42] TPC. 2000. TPC Benchmark W (Web Commerce). http://www.tpc.org/tpcw. (2000).

[43] Liang Huai Yang, Mong Li Lee, and Wynne Hsu. 2003. Efficient Mining of XML Query Patterns for Caching. In *PVLDB (VLDB '03)*. VLDB Endowment, 69–80. http://dl.acm.org/citation.cfm?id=1315451.1315459

[44] Barzan Mozafari Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella. 2017. Database Learning: Toward a Database that Becomes Smarter Every Time. *SIGMOD* (2017), 587–602.

[45] E. E. Yusufoglu, M. Ayyildiz, and E. Gul. 2014. Neural network-based approaches for predicting query response times. In *2014 International Conference on Data Science and Advanced Analytics (DSAA)*. 491–497. https://doi.org/10.1109/DSAA.2014.7058117