ChronoCache: Predictive and Adaptive Mid-Tier Query Result Caching

Brad Glasbergen, Kyle Langendoen, Michael Abebe, Khuzaima Daudjee
Cheriton School of Computer Science,
University of Waterloo
{bjglasbe,kjlangen,mtabebe,kdaudjee}@uwaterloo.ca

ABSTRACT

The performance of data-driven, web-scale client applications is sensitive to access latency. To address this concern, enterprises strive to cache data on edge nodes that are closer to users, thereby avoiding expensive round-trips to remote data centers. However, these geo-distributed approaches are limited to caching static data. In this paper we present ChronoCache, a mid-tier caching system that exploits the presence of geo-distributed edge nodes to cache database query results closer to users. ChronoCache transparently learns and leverages client application access patterns to predictively combine query requests and cache their results ahead of time, thereby reducing costly round-trips to the remote database. We show that ChronoCache reduces query response times by up to 2/3 over prior approaches on multiple representative benchmark workloads.

CCS CONCEPTS

• **Information systems → Middleware for databases**.

KEYWORDS

predictive caching; prefetching; distributed data management

ACM Reference Format:
Brad Glasbergen, Kyle Langendoen, Michael Abebe, Khuzaima Daudjee. 2020. ChronoCache: Predictive and Adaptive Mid-Tier Query Result Caching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3380593

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380593

1 INTRODUCTION

Data-driven applications continue to rise in popularity across numerous domains, and databases are the primary means of storing and querying their data [40]. Applications interact with databases according to a client-server model; the application (client) composes a request and issues it to the remote database machine (server), after which the database server processes the query and returns its response. Thus, *each request* incurs latency overheads due to network round trip times. It is well known that these communication overheads are considerable [8, 26], and are compounded as applications become increasingly geo-distributed to serve worldwide clients.

Consider the queries in Figure 1a, which are taken from the TPC-E benchmark's Market-Watch transaction [15]. The first query retrieves a set of market symbols, and subsequent queries loop over each symbol to retrieve the number of outstanding shares. As the first query retrieves an average of 100 rows, dozens of subsequent queries are issued to the remote database. Each of these remote requests incur costly network round trips and greatly increase the latency of the overall transaction.

The number of remote requests, and hence expensive network round-trips, can be reduced by combining these separately issued queries together into a single query that retrieves their combined result sets (Figure 1b). For example, a client could issue the entire transaction as a stored procedure, or they could obtain a combined result for the queries using a join. Combining the queries results in a tremendous reduction in latency — our experiments show that intelligently combining queries in a trans-continental wide-area network (WAN) setting cuts the average query response time of the entire TPC-E workload *by nearly 2/3* – but comes with a substantial downside: the application developer must manually inspect the application's code and replace queries amenable to these optimizations with more complicated combined queries or stored procedure logic. Furthermore, combining queries presents complex software engineering challenges [34]: a combined query or stored procedure is less composable than individual queries, and more difficult to update when the application changes.

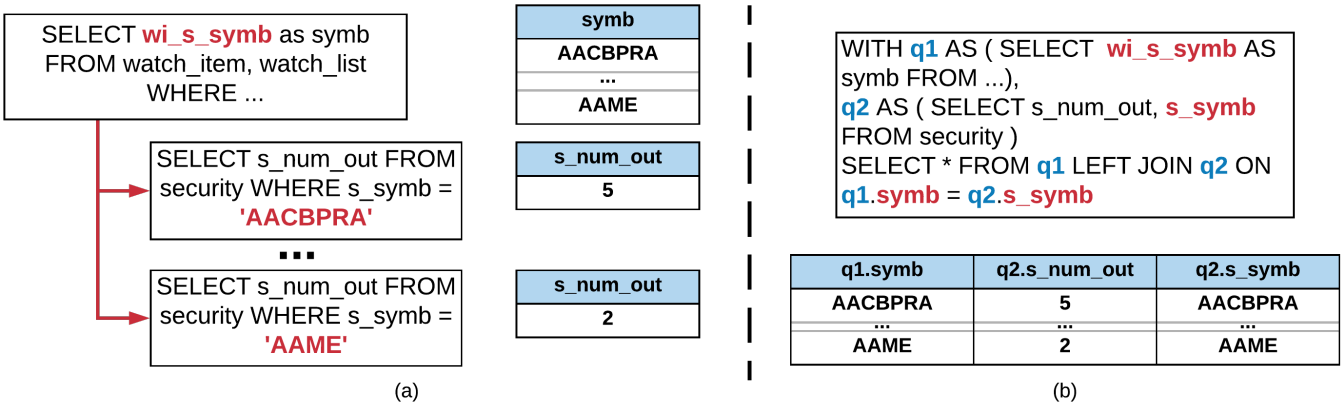


Figure 1: An example of ChronoCache exploiting query patterns by combining queries. The boxes with SQL text represent submitted queries; the associated result sets are adjacent. (a) shows the original query texts and (b) shows the combined query.

Many workloads contain query patterns similar to the TPC-E example that can be exploited for response time reduction [9, 15, 18, 22]. In this paper, we present **ChronoCache**, a shared middleware caching layer that automatically discovers patterns among client application queries, exploiting them to predictively cache query results ahead of time and minimize query response times. Importantly, ChronoCache obviates the need for manual code inspection and does not modify application logic: clients obtain these response time reductions simply by submitting queries to ChronoCache instead of directly to the database.

ChronoCache models client workloads, learning patterns among queries and exploiting them by combining queries into a single request that captures all of their result sets. Afterward, ChronoCache decodes the combined result into result sets for each of the original queries and caches them on edge nodes near clients ahead of time. In doing so, the client avoids costly round trips to a remote database for all of these queries and thus experiences large reductions in response time. Importantly, ChronoCache performs these optimizations transparently and in an online fashion — clients interacting with ChronoCache need not be aware of its query combination optimizations and ChronoCache itself requires *no offline training or static analysis*. ChronoCache shares cached results among clients using intuitive consistency semantics and supports a wide range of common query patterns. These design decisions ensure that ChronoCache scales well to serve clients in a geo-distributed setting.

Detecting queries that would benefit from being combined, determining an appropriate combination strategy that preserves the semantics of the original queries, and decoding the combined result set are challenging objectives. Performing these tasks while the system is executing, with information available only at the middleware layer, and without any offline training further increases their difficulty. In this paper, we present how ChronoCache meets all of these challenges by addressing the following questions:

- How can we detect queries that should be predictively combined *while the system is executing?*
- How should we model query relationships to exploit optimization opportunities while avoiding predictive query redundancy?
- How can we combine queries in a way that preserves the semantics of the original queries while supporting a broad class of query patterns?
- How do we effectively maintain and share cache results between clients with intuitive consistency semantics?

In the next section, we describe how ChronoCache detects query combination and prefetching opportunities with only the information available in a middleware caching layer. Section 3 describes how ChronoCache stores and manages discovered combination opportunities, with a focus on avoiding redundant predictive query combinations. Section 4 presents two strategies for combining queries into a single request and then decoding their result sets into that of the original queries, all while retaining their original semantics. Section 5 discusses system details of ChronoCache, and Section 6 empirically demonstrates ChronoCache's superiority over existing approaches. We discuss related work in Section 7 and conclude in Section 8.

2 DETECTING QUERY RELATIONSHIPS

ChronoCache (Figure 2) monitors client query submissions and builds models that capture relationships among queries through structures that we call *dependency graphs*. When a query arrives, ChronoCache uses these dependency graphs

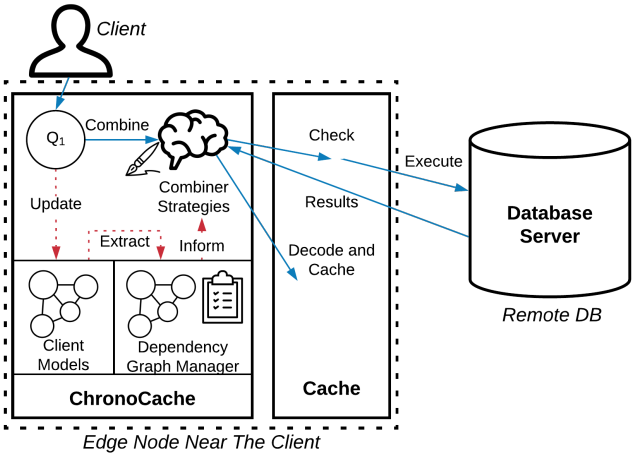


Figure 2: ChronoCache's system architecture.

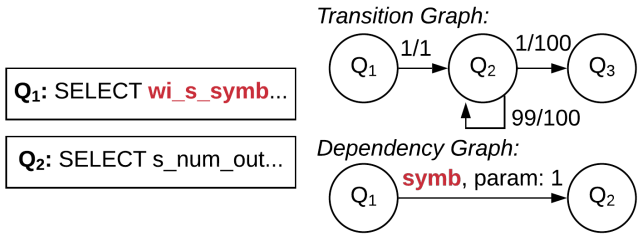


Figure 3: A client's query transition graph and dependency graph for the queries shown in Figure 1.

to determine if the query should be combined with a set of predicted upcoming queries (Section 3). If so, ChronoCache predictively combines the queries using one of its *combination strategies* (Section 4), after which the combined query is submitted to the database server. The server then executes the combined query and returns its result set. ChronoCache splits this result set into multiple result sets, one for the original query and one for each of the predicted upcoming queries (Section 4.1.1), caching these results on an edge node near the client ahead of time. Asynchronously, ChronoCache updates models that describe client application query submission behaviour and extracts query relationships from them on-the-fly in the form of dependency graphs to power future predictive caching.

ChronoCache subsumes Apollo's [22] monitoring and query relationship extraction components, building on them by supporting nested/loop queries and complex query dependency hierarchies. These components view a client's workload as a stream of query submissions $Q_1, Q_2, \ldots$ to the remote database. This stream of queries is encoded into a *query transition graph* [22], a model that describes client query behaviour and enables ChronoCache to predict which queries a client is likely to execute next (Figure 3). The nodes in the transition graph correspond to constant-agnostic representations of queries called *query templates* [22]. An edge between two nodes (Q_1, Q_2) is labelled with a probability denoting the likelihood that Q_2 will be submitted by the client within a small time interval Δt of Q_1 (Figure 3). When the probability of executing Q_2 within Δt of Q_1 exceeds a predefined threshold parameter τ, we say that Q_2 is *temporally correlated* to Q_1. ChronoCache constructs the query transition graph entirely while the system is online. At the same time, ChronoCache examines the graph's structure to uncover relationships among a client's queries.

In the following sections, we simplify our explanations by describing how ChronoCache extracts and exploits query relationships for a single client. In practice, ChronoCache discovers relationships and predictively caches query results in parallel for concurrent clients.

2.1 Discovering Query Patterns

ChronoCache uses a client's query transition graph to model temporal query relationships and predict which queries the client is likely to execute next. To demonstrate this capability, consider the query transition graph in Figure 3, which captures the relationships among the queries from Figure 1. Recall that Q_1 executes and returns 100 rows, so Q_2 subsequently executes 100 times — once for each row in Q_1's result set. After executing all of these loop queries, the client executes a query Q_3. Because Q_2 always executes after Q_1, the edge from Q_1 to Q_2 has probability 1. There are 99 transitions from Q_2 to Q_2 out of its 100 executions, so Q_2 has a self-edge labelled with probability 99/100 and an edge to Q_3 labelled with probability 1/100. Using its enhanced transition graph, ChronoCache predicts that Q_2 is likely to execute after Q_1 and that Q_2 will execute multiple times in a row.

High transition probabilities indicate strong relationships among queries, suggesting that they may be good candidates for query combination. Thus, once ChronoCache has determined that a set of queries is temporally correlated, it looks for data dependencies between them. Concretely, for the relationship between Q_1 and Q_2, it considers whether Q_1's result set contains values that are used as input parameters for Q_2. If all of Q_2's parameters can be determined from Q_1, then ChronoCache will also retrieve Q_2's result set whenever the client issues Q_1.

ChronoCache determines whether a query Q_i's result set contains values used as input parameters for a query Q_j by recording the last result set returned for each query template, as in prior work [9, 22]. If Q_j is temporally correlated to Q_i, then ChronoCache checks the rows in Q_i's result set to determine if any column's value matches an input parameter of Q_j. ChronoCache records each matching value from Q_i's result set as a mapping from a column in Q_i's select list to

an input parameter in Q_j. For example, consider the queries from Figure 1: the symb column's value in the first row of Q_1's result set matches the first input parameter of the first iteration of Q_2. If all of Q_j's input parameters can be determined from this row in Q_i's result set, then ChronoCache records that Q_i should be combined with Q_j. A client may issue Q_j again before issuing Q_i, in which case ChronoCache looks for mappings in the next row of Q_i: this corresponds to a loop structure in which an instance of Q_j is executed for each row in Q_i (as in Figure 1).

The validity of these *parameter mappings* is re-evaluated each time ChronoCache observes a set of queries matching a query pattern [22]. If a mapping does not hold, then ChronoCache deems it spurious because an input parameter's value coincidentally matched an output column value; the mapping is blacklisted and never used in the future.

In practice, multiple queries may provide parameter mappings for an upcoming query. To accommodate these relationships, ChronoCache considers whether *any* queries that are temporally correlated with Q_j provide parameter mappings. Each query providing input parameters to Q_j may in turn have its parameters provided by another set of prior queries, forming a hierarchical structure of parameter sharing relationships. ChronoCache uncovers these structures by applying the above technique recursively, synthesizing the hierarchical nature of parameter sharing into *dependency graphs*.

2.1.1 Representing Query Patterns. Dependency graphs describe the dependency-like nature of parameter sharing among queries. A directed edge in a dependency graph from a node for Q_i to a node for Q_j indicates that Q_i supplies an input parameter value for Q_j. This directed edge contains information describing which column(s) from Q_i's result set maps to which parameter position(s) in Q_j. For the queries from Figure 1, there is a directed edge from Q_1 to Q_2 and a mapping from the symb field to the first parameter position in Q_2 (Figure 3). The dependency graph tells ChronoCache when and how the involved queries should be combined; in this example, ChronoCache determines that Q_2's first input parameter relies on the values in the symb column in Q_1's result set, and thus predictively combines Q_1 with Q_2 and executes them as one request whenever the client submits Q_1.

Constructed dependency graphs are passed to ChronoCache's dependency graph manager and predictive combiner modules, which exploit them to decide when and how to combine queries for predictive caching (Sections 3 and 4).

2.2 Discovering Complex Loops

Thus far, we have considered combining Q_j with its prior queries only if those prior queries' result sets determine all of Q_j's parameters. However, loop query-patterns often contain

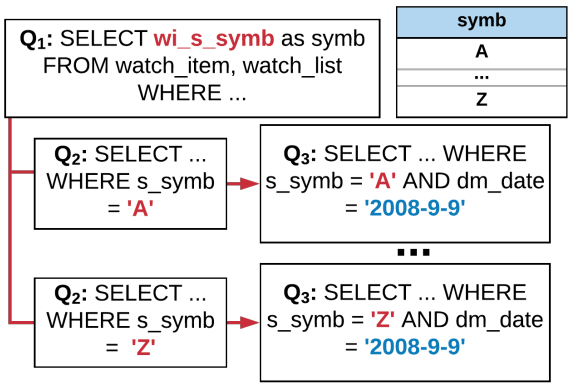


Figure 4: An example of a per-loop fixed constant (dm_date) that cannot be determined from the result set of a prior query.

constants that are not part of these result sets. For example, we simplified the TPC-E benchmark's Market-Watch transaction in Figure 1; in reality, the loop consists of multiple queries, some of which rely on a *per-loop* constant not contained in any of the queries' result sets (Figure 4) such as the *dm_date* predicate. The value for this predicate varies per loop invocation, but is constant within a loop. Consequently, the above approach would not detect these queries in the loop as candidates for combination. It is imperative, however, that these queries are detected and optimized: for every such query in a loop iterating over a result set of size N, ChronoCache's combination strategies *eliminate* $N-1$ network round trips.

ChronoCache analyzes a client's query transition graphs to detect and extract loops of queries. We present the intuition behind this extraction process using the query transition graph (Figure 5) for the loop queries in Figure 4. Note that if the loop has N iterations, then Q_2 will transition N times to Q_3 and Q_3 will transition $N-1$ times to Q_2. As N grows, the transition probability among the queries in the loop (Q_2 and Q_3) approaches the value 1. Consequently, there is a directed edge from Q_2 to Q_3, as well as a directed edge from Q_3 to Q_2, both with probability greater than the temporal correlation threshold τ for a sufficiently large N. In other words, Q_2 and Q_3 form a *strongly connected component* over temporally correlated queries in the graph. As such, ChronoCache can identify loop structures from the query transition graph by searching for strongly-connected components over edges between temporally correlated queries. Mathematically, loops will remain if they have more than $\lceil \frac{1}{1-\tau} \rceil$ iterations.[1]

[1] In our experiments, we use $\tau = 0.8$, which means ChronoCache finds loops with more than 5 iterations. The sensitivity of τ is discussed in Section 6.7.

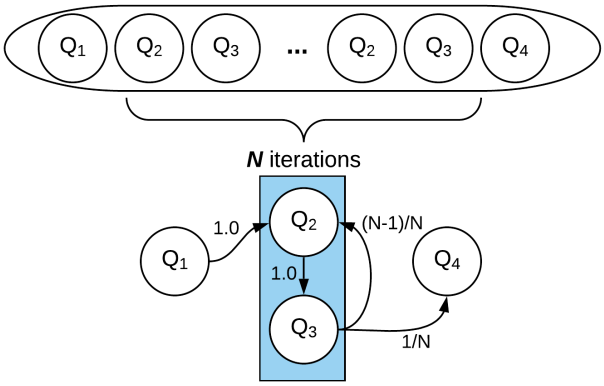


Figure 5: A stream of loop queries translated into a query transition graph.

ChronoCache employs this strongly-connected components strategy to find loops. In particular, for a query Q_j under consideration, it relaxes the parameter mapping constraint from the prior section and searches for strongly-connected components containing Q_j's node in the τ-pruned query transition graph. To find these components efficiently, ChronoCache uses Tarjan's algorithm [41]. ChronoCache verifies that the extracted components conform to true loops by checking if each node *within* a component relies on a parameter mapping from some *source query* outside of the component, corresponding to the query over whose result set the loop is iterating. In the example in Figure 5, the source query is Q_1.

ChronoCache encodes extracted loops and their source queries into dependency graphs, as in the previous section, and passes the dependency graphs to the dependency graph manager module (Section 3). However, it marks queries in the loop that contain per-loop constants. These marked queries are used by ChronoCache's dependency manager and query combination strategies to indicate that ChronoCache should wait for at least one iteration of the loop before trying to optimize it by combining the loop's queries. In doing so, constants not supplied by parameter mappings will become known through client-supplied query texts for the first iteration of the loop. For example, in Figure 4, ChronoCache will wait for the first instance of Q_3 to observe the *dm_date* predicate constant, before optimizing the combined loop queries. As before, loop structures can be composed into hierarchies of loops to enable more complex predictive caching.

ChronoCache's loop detection techniques may find dependency graphs discoverable by the simpler detection method, resulting in duplicates. Moreover, some of the extracted dependency graphs may subsume others; that is, one dependency graph includes all of the information available in another. Next, we describe how ChronoCache addresses these challenges by efficiently managing and exploiting these dependency graphs for predictive caching, thereby avoiding predictive optimization redundancy and reducing load on the remote database server.

3 DEPENDENCY GRAPH MANAGEMENT

ChronoCache inspects dependency graphs to determine *i*) when a dependency graph is ready to be used for predictive caching, and *ii*) which dependency graphs are redundant and should be precluded from use.

Algorithm 1 ChronoCache Query Processing

```
Require: Incoming query Q_i for client c
1:  dep_table = get_client_dep_table(c)
2:  ready_graphs = dep_table.mark_text_avail(Q_i)
3:  // In parallel:
4:  for graph ∈ ready_graphs do
5:    new_text = combiners.optimize(graph)
6:    rs = send_to_db_and_cache(new_text, graph)
7:    dep_table.split_mark_text_avail(Q_i, rs)
8:  end for
9:  if ready_graphs.empty() then
10:   rs = send_to_db_and_cache(get_text(Q_i))
11: end if
```

ChronoCache stores extracted dependency graphs in a per-client *dependency table* that is used to determine when the queries in the dependency graph are ready to be predictively combined, executed, and have their results cached. An entry in the dependency table is a mapping from a query to a dependency graph, corresponding to a query whose text must be available before the dependency graph's queries can be predictively executed. To determine these *dependency queries*, ChronoCache simply checks which queries in the dependency graph have input parameters that cannot be determined from other queries in the dependency graph. In our running example from Figure 3, Q_1 is the sole dependency of the dependency graph.

When the last dependency query Q_i arrives for a dependency graph (Algorithm 1, line 2), then the graph is satisfied. ChronoCache forwards a satisfied dependency graph and the query Q_i to the query combiner (Section 4), which optimizes the query by combining it with other queries in the dependency graph given parameter mappings and previously executed queries' texts (line 5). If Q_i is not the last dependency query to arrive for any dependency graph, then ChronoCache submits the query to the database unaltered, and caches the query result set for future requests (line 10). If Q_i's query text is needed for parameter mappings to other queries, then it is stored for use during query combination.

As previously mentioned, some dependency graphs may subsume others. For example, in Figure 6, dependency graph

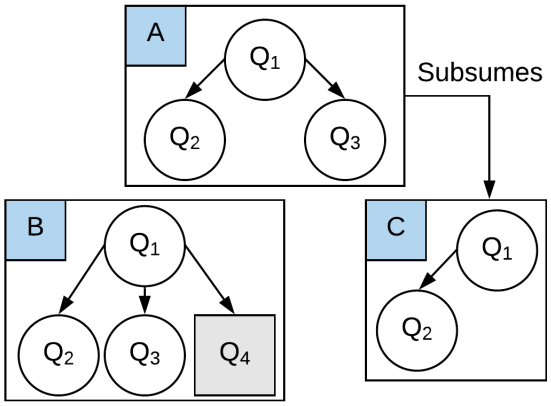


Figure 6: An example of dependency graph relationships.

A contains all the same parameter mappings as dependency graph C, in addition to a mapping from Q_1 to Q_3. If ChronoCache retains both graphs in the dependency table, then both will become ready for predictive caching when Q_1 arrives. Because a combination of the queries in A retrieves a result set that is a superset of the combination of queries' result set in C, retaining both graphs leads to redundant optimization overhead and additional database queries. To avoid this redundancy, ChronoCache removes dependency graph C from consideration for predictive optimization. However, graphs that contain a loop-constant dependency are not considered supersets of graphs without such a dependency, because loop-constant relationships force ChronoCache to wait for an iteration of the loop before predictively executing the queries in the graph (Section 2.2). Therefore, neither A nor B is a superset of the other, and the client's dependency table will retain dependency graphs A and B but not C.

To retain only these superset dependency graphs, ChronoCache uses the following merge procedure when it has extracted a dependency graph g from a client's query transition graph. If the dependency graph has already been added to the client's dependency table, then ChronoCache discards the dependency graph without further processing. Otherwise, ChronoCache searches the table for dependency graphs that contain any of the nodes in g, checks to see if they subsume g or vice versa, and if so merges the graphs together. If none of the graphs subsume the dependency graph (or vice versa), then ChronoCache adds the new dependency graph unchanged to the table.

4 QUERY COMBINATION STRATEGIES

When ChronoCache receives a query from a client, it checks to see if the query is the last dependency query for a dependency graph. If so, Chronocache employs one of the following two strategies to generate a query that combines all of the queries in the dependency graph together, predictively executes the combined query, splits the returned result into result sets for each of the queries in the dependency graph, and caches them. ChronoCache decides which strategy to use based on characteristics of the queries in the dependency graph.

The first strategy, which uses left joins over common table expressions, is superior for select-project-join queries with simple filtering conditions. The second strategy is effective for a broader class of queries, but results in more complex SQL query text and is thus slower to generate. We now describe these approaches.

4.1 Left Joins over Common Table Expressions

For dependency graphs that contain select-project-join queries without ordering conditions or aggregate clauses, ChronoCache builds on strategies that left-join the queries together using the parameter mappings between them [9].

Given a ready dependency graph g that contains predicted queries, ChronoCache uses Algorithm 2 to generate the combined query for predictive execution and caching. We use the queries from our running example to illustrate the process in Figure 7. First, ChronoCache topologically sorts the dependency graph to get an ordered list of queries (line 1), shown as Q_1, Q_2 in box 1 in Figure 7. This topologically ordered list indicates the join order for queries in the combined query. Next, ChronoCache writes out each query in topological order as a common table expression (CTE) (lines 10-20), adding a candidate key to the select list of each query to uniquely identify its rows (line 15), as shown in boxes 2 and 3 of Figure 7. ChronoCache forms this candidate key by concatenating row identifiers from each of the tables that the query accesses. ChronoCache strips filter conditions from each CTE that contain parameter mappings from other queries (line 14), e.g., the filter condition in Q_2 on `s_symb` is extracted and stripped. Finally, ChronoCache writes out a query that left-joins all of the CTEs together in topological order using the stripped-out conditions as join conditions (line 21-27), as in box 4.

This combined query is then executed against the database, returning a result set that contains the results necessary for each query in the dependency graph. Next, ChronoCache splits the combined query's result set into result sets for each of the queries in the dependency graph.

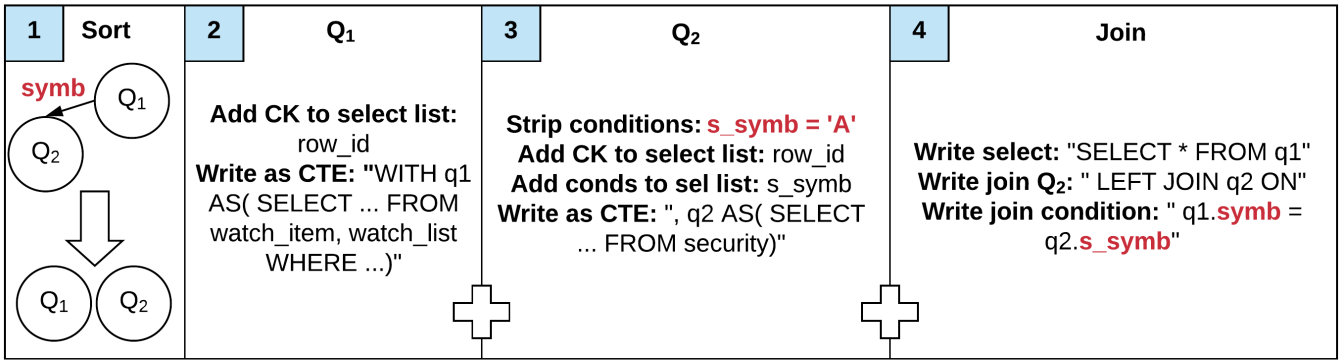


Figure 7: ChronoCache's CTE-join procedure for a dependency graph. ChronoCache topologically sorts the queries in the dependency graph, writes each query as a CTE, then left-joins the queries together on their shared parameters.

Algorithm 2 CTE-Join Strategy

Require: A ready dependency graph g

```
ordered_queries = topological_sort(g)
combined_text = ""
first_query = ordered_queries.pop()
text = add_candidate_key(first_query)
combined_text += "WITH q1 AS ("
combined_text += text
combined_text += ")"
join_conds = []
i=2
while !ordered_queries.empty() do
  q = ordered_queries.pop()
  uconds = q.uncontained_conds()
  join_conds.append(uconds)
  text = q.strip_uncontained_conds()
  text = add_candidate_key(q, text)
  text = add_sel_fields(uconds, text)
  combined_text += ", q" + i + " AS ("
  combined_text += text + ")"
  i += 1
end while
combined_text += "SELECT * FROM q1 "
for j = 2; j < i; j++ do
  combined_text += "LEFT JOIN q" + j
  combined_text += " ON "
  c = write_join_conds(join_conds[j-2])
  combined_text += c + " "
end for
```

4.1.1 Decoding Result Sets. After executing the combined query, ChronoCache receives a result set that contains all of the values that would have been returned by the original queries in the dependency graph. ChronoCache iterates over the combined result set and splits it into result sets that match the result sets of sequential non-predictive query execution. These result sets are then cached. Thus, when the client submits the queries predicted by ChronoCache, the results are returned from the local cache instead of using costly remote database accesses.

Q1_ROW_ID	SYMB	Q2_ROW_ID	NUM_OUT	SYMB
1	'A'	1	20	'A'
2	'B'	2	5	'B'
2	'B'	3	10	'B'
3	'B'	2	5	'B'
3	'B'	3	10	'B'

Figure 8: ChronoCache splitting a CTE-joined query's result set into that of the original queries.

We describe how ChronoCache splits CTE-combined result sets using the combined result set from Figure 8. First, ChronoCache creates an empty result set for each query in the dependency graph as well as a list of result sets for each query. The former is used as a running result set to which ChronoCache adds rows as it iterates through the combined result set, and the latter contains completed result sets corresponding to each iteration of each of the queries.

ChronoCache iterates over each row and splits it into result sets for the original queries. For a given row, ChronoCache determines which column-values belong to each query using the select-lists of the original queries. For example, in Figure 8, the `symb` field belongs to Q_1 and `num_out` belongs to Q_2. ChronoCache adds these values as rows to the ongoing result sets for each query. ChronoCache now considers the next row, and determines the values for each row in the same way. If any of the queries on which a query depends have a changed candidate key, then the row should be part of a new result set, and the old result set should be added

to the list of finished result sets. In the example in Figure 8, Q_1 does not have any queries on which it depends, so ChronoCache adds the extracted row for Q_1 to the ongoing result set, which now contains 2 rows. Because Q_2 depends on Q_1 and Q_1's candidate key has changed, ChronoCache determines that the extracted row for Q_2 forms a new result set corresponding to the iteration of Q_2 defined by the new row from Q_1. Thus, ChronoCache adds the old result set for Q_2 to the list of completed result sets, and adds the extracted row for Q_2 to a new result set.

ChronoCache uses candidate keys to avoid adding duplicate rows to a result set. These duplicate rows can occur because the left join for a row from Q_1 matches multiple rows in Q_2. For example, ChronoCache does not add Q_1's extracted row for row 3 to its ongoing result set because the candidate key for Q_1 is the same in rows 2 and 3 (Figure 8). By contrast, ChronoCache does add the fourth row, which contains the same symbol, to the ongoing result set for Q_1 because it has a different candidate key. Upon completion, ChronoCache returns a result set for Q_1 containing the three blue rows in Figure 8. ChronoCache returns three result sets for Q_2 corresponding to each iteration of Q_2 executed using parameters from Q_1. These result sets are shown in the non-blue colours in the figure.

Once ChronoCache has constructed the result sets for each query in the dependency graph, it caches the results. Cached result sets are keyed by the string of the query that would have generated them. For example, the single result set for Q_1 is keyed by Q_1's text, and the i^{th} result set for Q_2 is keyed by the text for Q_2 parameterized by the i^{th} row of Q_1's result set.

4.2 Lateral Union

Queries with aggregates, ordering conditions, limit clauses, and other advanced SQL query constructs cannot be handled using the CTE-join strategy and are instead handled using a lateral union strategy. Although the lateral union strategy can combine queries optimizable via the CTE-join, it generates more complex SQL queries. Thus, ChronoCache uses the CTE-join strategy wherever possible, falling back to the lateral union strategy only when necessary.

ChronoCache builds on a lateral union strategy for combining queries from previous work [9], but addresses two limitations. The lateral union strategy described in [9] cannot be used to combine queries where multiple prior results determine the result set of a subsequent query, unless the prior queries themselves have no dependencies. E.g., for four queries Q_1, Q_2, Q_3, Q_4, where Q_1's result set determines Q_2 and Q_3's input parameters, and Q_2 and Q_3 in turn define Q_4's input parameters, the strategy cannot be applied. Fundamentally, this problem occurs when there are multiple queries at the same *topological height* — that is, if we topologically sort the dependency graph for $Q_1, \ldots, Q_4$, both Q_2 and Q_3 have the same path length to Q_4. Q_2 and Q_3 do not share parameters, but both of their result sets are used by Q_4. The second limitation is that this lateral union strategy requires that each relation have a defined and known candidate key.

ChronoCache overcomes these limitations by inducing its own candidate keys on intermediate result sets and joining queries at the same topological depth by their row number. In particular, ChronoCache uses the `ROW_NUMBER()` generic SQL window function to assign a row number to each row in a query's result set, which acts as a candidate key for that result set. When ChronoCache detects that there are multiple queries at the same topological height, it transforms the dependency graph by combining queries at the same height together via a join on their `ROW_NUMBER()`. After applying this process such that every topological level has exactly one query, it employs the lateral union strategy [9]. ChronoCache iterates over each query in the transformed dependency graph and writes them as a lateral derived table over their dependency queries, thereafter selecting all the necessary fields from each of these tables to form the combined query. Once ChronoCache has received a result set for a query combined using this strategy, it employs a split algorithm similar to the one described for CTE-combined queries [9].

5 THE CHRONOCACHE SYSTEM

We implemented ChronoCache as a middleware caching layer between application clients and the remote database. Clients issue queries to ChronoCache, which then interacts with the cache and remote database on the client's behalf to return a result set. ChronoCache uses Memcached [1], a popular industrial caching solution, as its query result cache. ChronoCache communicates with a remote PostgreSQL database [23] using JDBC, though any ANSI SQL compliant [6] relational database management system may be used instead of PostgreSQL.

When a client submits a query to ChronoCache, ChronoCache parses the query using the PlSql grammar [2] for the ANTLR 4 parser [33]. After parsing the query, ChronoCache builds a constant-agnostic version of the query's parse tree to identify its query template, storing a copy of this parse tree for later use. Next, ChronoCache uses the query template's identifier to check if it is the last dependency query for any of the client's dependency graphs. If so, ChronoCache retrieves the parse trees for each of the queries in the dependency graph, and combines them using the CTE-join (Section 4.1) or Lateral Union (Section 4.2) strategy. The raw query text output from the query combiner is then forwarded to PostgreSQL using JDBC, which returns a result set for the combined query. ChronoCache splits this result into result

sets for each of the queries in the dependency graph, considering their parse trees to determine which fields in the result set are relevant to each query (Section 4.1.1). Finally, ChronoCache caches all of these result sets in Memcached and returns to the client the result set corresponding to the original query.

ChronoCache exploits asynchronous execution to mitigate query processing overheads. When the queries in multiple dependency graphs become simultaneously ready for predictive execution, ChronoCache will combine their queries and cache their results in parallel. As these result sets are cached, more dependency graphs may become ready as a consequence of retrieving these new result sets. The queries in these dependency graphs will be predictively combined and cached as a background process, thereby minimizing the overheads on client-submitted queries.

Although ChronoCache could predictively execute queries that update the database state, doing so results in considerable overheads to account for possibly incorrect predictions. If an update query is incorrectly predicted, it must be rolled back. Also, to ensure that clients do not observe updates from an incorrectly predicted update, ChronoCache would need to preclude clients from viewing the results of any predicted update query until it is certain that its prediction is correct. Given the overheads involved in managing these requirements, ChronoCache focuses on predictively caching read queries.

Because ChronoCache's query combinations are *predictive*, it is possible for ChronoCache to incorrectly predict that a query Q_2 will follow the current query Q_1 and submit a combined query on these queries' behalf. However, doing so does not affect the correctness of applications interacting with ChronoCache. As submitted queries, and not predicted queries, are used to update ChronoCache's predictive model, the probability of Q_2 following Q_1 will be adjusted downward accordingly to avoid mispredicting in the future.

Conditional logic in client applications affects the probability of queries following each other. For example, Q_1 may be followed by Q_2 with probability 0.8 and Q_3 with probability 0.2 based on an `if` condition in the application. As ChronoCache operates in middleware, program source code is unavailable and cannot be exploited to determine with certainty when queries will follow one another. However, the τ threshold parameter ensures that Q_1 is combined with its subsequent queries only if they follow Q_1 with probability at least τ. Therefore, an appropriate value of τ trades-off between increasing load on the database by combining queries from multiple branches in exchange for more cache hits, or vice versa. Consequently, ChronoCache remains effective on workloads with many conditional statements.

5.1 Avoiding Redundant Predictions

When ChronoCache receives a query from a client, it checks to see if a dependency graph is ready to be predictively executed and cached. However, if the predicted queries' result sets are already cached, then combining the queries in the graph, executing the combined query, and caching the split queries is redundant. Moreover, doing so will introduce query processing overheads and hamper performance.

To avoid this redundant work, ChronoCache determines if result sets are available for each of the predicted upcoming queries. Using the queries from Figure 1, if a result set for Q_1 is available, then ChronoCache verifies that a result set for each of the Q_2 queries generated by the rows in the cached result of Q_1 are also available in the cache. If so, ChronoCache does not combine the queries, and submits Q_1 unchanged, which results in a cache hit.

Multiple clients may also wish to execute the same (possibly combined) read query against the remote database simultaneously. In such cases, ChronoCache will submit the query only once, and block all other clients executing the same query until the result set is returned [22]. The result set is then forwarded to the waiting clients. By employing this procedure, ChronoCache further reduces the load on the remote database.

5.2 Session Semantics

Sharing cached results among clients can provide substantial performance gains (Section 6). However, a naïve implementation of shared cached results may introduce undesirable effects, like a client retrieving a stale result set from the cache for relations it has since updated. Thus, cached results in ChronoCache are shared based on session semantics [16] over client query submissions: using strictly serializable databases and when query result return order represents commit order, the caching scheme ensures that a client never observes a result set that corresponds to a database state older than what it had accessed before. Otherwise, returned results correspond to consistent snapshots.

To provide consistent session semantics, ChronoCache locally maintains a version vector V_d indicating the database's state. An entry in the vector, $V_d[i]$, corresponds to the observed version of a relation R_i; thus, V_d has dimension equal to the number of relations in the database schema. Initially, the version of each relation in V_d is 1. When a client updates a relation R_i, ChronoCache atomically increments $V_d[i]$.

Each client c has its own session, and ChronoCache maintains a version vector V_c for each client indicating the version of the relations that it has most recently accessed. When a client submits a read query to the database, ChronoCache sets $V_c = V_d$. The result set for that query is tagged with the client's version and cached. A result set in the cache can only

be used by a client if the result set's version is greater than or equal to the client's version. Concretely, for all relations R_i accessed in the associated query, Chronocache ensures that $V_r[i] \geq V_c[i]$, where V_r is the cached result's version vector. Upon accessing the cached result, ChronoCache sets $\forall i : V_c[i] = max(V_c[i], V_r[i])$.

The above approach provides session semantics for a single-node ChronoCache deployment. In a multi-node deployment, the database state may advance without each ChronoCache instance updating its database version vector V_d, as these vectors are maintained locally. To remedy this problem, ChronoCache instances can be configured to always increment each index of V_d whenever a client accesses the remote database, rather than relying on the node's stored value of V_d to encapsulate all of the previous updates. Cached result sets must then additionally be keyed by a ChronoCache node identifier to preclude sharing results across nodes as version vectors cannot be compared across nodes.

5.2.1 Access Control. It is not always appropriate to share cached query results between clients. Access control schemes such as row-level security may cause a query issued by one client to return different results than when it is executed by another. Sharing results between such clients would violate security policies. These considerations can be addressed through a straightforward extension of the version vector access mechanisms.

Each client is grouped with other clients according to their security policies. In particular, a client c_i is in the same group SG_k as c_j if and only if they have the same security policies and thus would return the same result set for any query Q on a fixed database state (snapshot). We assign an integer identifier k to each security group SG_k and additionally tag the version vectors for cached results and clients to include the security group identifier. When reading a result from the cache, a client must ensure that its security group identifier matches that of the cached result in addition to meeting the other requirements in Section 5.2. Doing so ensures that each client sees a result only if it does not violate security requirements. Even in cases where every client has its own set of security rules, predictive caching results in considerable performance improvements, as we show in the subsequent section.

6 PERFORMANCE EVALUATION

In this section, we empirically evaluate the effectiveness of the ChronoCache system at reducing end-to-end query response times as a geo-distributed edge caching system. By default, each experiment uses four machines: a benchmark machine that drives the workload, the query result cache (Memcached 1.5.14 [1]), the ChronoCache (or comparative) system, and the remote database (PostgreSQL 12.1 [23]). The benchmark, ChronoCache, and database are each deployed on m4.4xlarge instances on Amazon AWS EC2, which have 16 virtual CPUs and 64 GB of RAM. The database instance is configured with a 200 GB SSD, while the other instances use a 50 GB SSD. The Memcached machine is deployed on a c3.large instance with 2 virtual CPUs and 4 GB of RAM. In all experiments, we allocate 3 GB of memory for caching.

Systems: We compare ChronoCache against the industry-standard LRU caching scheme and two state-of-the-art predictive caching techniques (Section 7). We implement these alternative designs within the ChronoCache system to ensure an apples-to-apples comparison.

- **LRU** is the industry-standard approach for query result caching. We configured ChronoCache to act as an LRU cache by retrieving results from Memcached if available and querying the remote database when necessary. In this mode, none of ChronoCache's modelling or predictive caching capabilities are used.
- **Apollo** is a predictive query caching system developed by Glasbergen et al. [22]. As Apollo cannot exploit loop query patterns or combine queries to reduce round trips, we configure ChronoCache not to recognize these types of patterns and submit predicted queries to the database sequentially, rather than using the combining strategies in Section 4.
- **Scalpel:** is a client-side prefetching tool [9] that requires a training period before its predictive system can be used, caches query results only within the scope of a transaction and without sharing results between clients. We remove these limitations from Scalpel by comparing against an augmented Scalpel in two ways: (i) Scalpel-E is an enhanced version that shares cached query results across transaction boundaries, and (ii) Scalpel-CC is an advantaged version enhanced with ChronoCache's full client query result sharing framework. Both systems are augmented with ChronoCache's adaptive learning framework to discover and model client query patterns.

Each of these systems is executed using the same client session consistency level (Section 5.2) and the same size cache. Before gathering results, we warmed up the database by executing the query workload for 20 minutes. Since we wanted to assess the effectiveness of ChronoCache's predictive caching, each of our experiments is initialized with an empty cache. In all experiments, each result is the average over five 5-minute runs. 95% confidence intervals are shown as error bars around each data point.

Workloads: We compare the systems using four popular workloads representative of geo-distributed applications:

- **TPC-E [15]** is a large, complex, online transaction processing workload (OLTP) that emulates the activities of a brokerage firm. The TPC-E schema defines 33 tables with 188 columns, and more than 50 unique query templates within 12 transactions [12], making it a challenging workload for ChronoCache's query pattern detection and predictive caching framework. We consider the full workload, except for a periodic data maintenance transaction to ensure consistency in our performance results. The database size is 70 GB per the workload's default scale factors. We drive the workload using DBT5 [3]. The TPC-E mix contains approximately 75% read-only transactions.
- **Wikipedia [18]** is a real Wikipedia workload based on a transaction and data distribution trace [18] containing the site's most popular transactions. We use 100,000 pages and 200,000 users resulting in a default database size of 22 GB and select page accesses according to a Zipf($\rho = 1$) distribution. We drive this workload using OLTPBench [18]. The Wikipedia workload contains 92% read-only transactions.
- **SEATS [18]** models an airline ticketing system where remote clients book reservations on flights. We configure the system with 80% read-only transactions and consider booking flights with up to 200 units of distance of their desired destination to increase workload complexity. We use an increased scale factor of 50 resulting in an initial database size of 12 GB. We drive this workload using OLTPBench [18].
- **AuctionMark [18]** emulates an online auction workload in which remote clients create auctions for items and bid on them. We configure the system with 85% read-only workload to represent a realistic auction workload as in [5], and such that each worker cooperatively closes auctions on average every 20 seconds (with 10 second variance). To demonstrate the importance of handling per-loop constants, we query the seller's average feedback rating over the last 30 days during the CloseAuctions transaction. On closing an auction, this useful feedback is attached to email correspondence between sellers and buyers [14, 19]. We increase the scale factor to 50 giving an initial database of size 12 GB. We drive this workload using OLTPBench [18].

Methodology: We start by considering the performance impact of ChronoCache on TPC-E query response times in its targeted WAN setting (Section 6.1). Afterward, we demonstrate a key feature of ChronoCache — its ability to learn over time (Section 6.2). Next, we evaluate ChronoCache's performance on the Wikipedia, SEATS and AuctionMark benchmarks (Sections 6.3-6.5). We present ChronoCache's scalability characteristics in Section 6.6. Finally, we conduct a sensitivity analysis of ChronoCache's configuration parameters (Section 6.7).

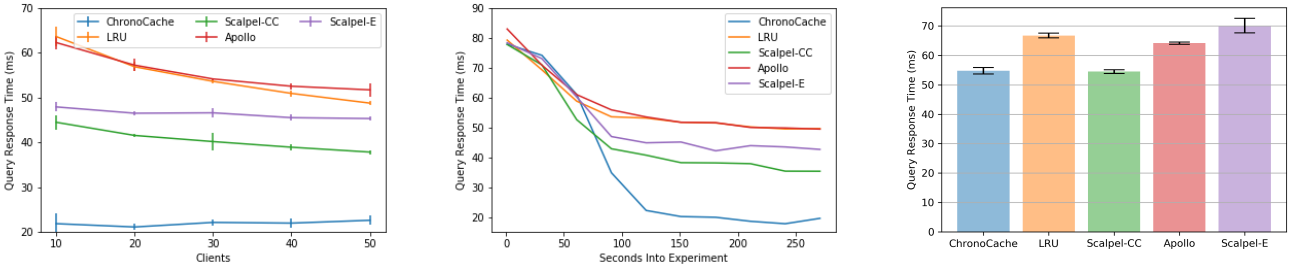
6.1 TPC-E Results

We evaluate ChronoCache's performance as a geo-distributed edge cache for dynamic data by deploying the benchmark machine, ChronoCache, and Memcached in the US-East region on AWS, and the database machine in the US-West region. Thus, the average round-trip time between the client machine and the database is 70 ms.

Figure 9a shows how the average query response time in milliseconds on the full TPC-E workload (for each system) varies while scaling the client load. ChronoCache reduces average query response time by nearly 2/3 compared to the LRU and Apollo systems, and by nearly 1/2 compared to Scalpel-CC and Scalpel-E. This large response time reduction is achieved by ChronoCache detecting and predictively caching a variety of query patterns ahead-of-time, resulting in up to 10x more cache hits than the other systems. In particular, ChronoCache maintains a 75% cache hit rate, while Scalpel-CC maintains 50%, Scalpel-E 45%, and LRU and Apollo 20% cache hit rates, respectively. Although Apollo has its own predictive caching capabilities, it does not support loop patterns like those shown in Figure 1 and submits its predicted queries *sequentially*. Therefore, Apollo does not significantly increase cache hits over LRU and does not reduce the number of round trips to the remote database. Scalpel-CC benefits from the online learning and shared caching semantics of the ChronoCache infrastructure, but is limited by its support for only a subset of the query patterns. In particular, it does not support loop queries with per-loop constants that are not contained in the result sets or input parameters of other queries, like those in the Market-Watch transaction (Figure 4). Thus, although Scalpel-CC improves performance over LRU due to its predictive caching, ChronoCache significantly outperforms it on the TPC-E workload. Scalpel-E supports the same patterns as Scalpel-CC, however its inability to share cached query results hampers performance. Each of the other systems experience a slight downward trend in response time due to these shared caching effects — with more clients executing and caching query results, there is a greater chance to experience a cache hit.

6.2 Learning over Time

To evaluate ChronoCache's online learning and adaptive capabilities, we measured average query response times over 30 second intervals for a TPC-E US-West experiment from Figure 9a. As Figure 9b demonstrates, ChronoCache rapidly



(a) TPC-E US-East to US-West **(b) TPC-E Learning Over Time** **(c) Wikipedia Results**

Figure 9: TPC-E and Wikipedia experiments for ChronoCache, LRU, Scalpel-CC, Apollo, and Scalpel-E. (a) depicts response times for the TPC-E workload (b) demonstrates system abilities to learn TPC-E query patterns, and (c) depicts response times for the Wikipedia workload.

learns client workload patterns. Within 150 seconds, ChronoCache has learned the important query patterns in the workload and has reduced response times to 25 ms, continuing to exploit these learned patterns thereafter effectively.

Because we have implemented Scalpel-CC and Scalpel-E on top of ChronoCache's client modelling and query pattern extraction framework, Scalpel-CC and Scalpel-E exhibit similar learning capabilities. However, Scalpel by default requires a training period [9] and does not learn while its predictive system is online. Despite being advantaged by ChronoCache's adaptive learning framework, neither Scalpel-CC's nor Scalpel-E's query response times converge to ChronoCache's because ChronoCache supports a broader class of query patterns.

Apollo's performance also improves over time, a consequence of its ability to learn from client workloads and share cached results among clients. However, Apollo does not support many of the types of query patterns in the TPC-E workload, and does not predictively combine queries as part of its caching techniques.

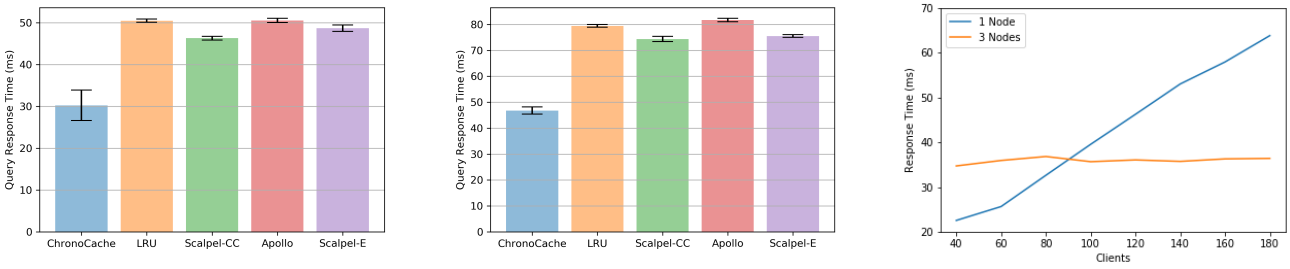
Finally, we observe that LRU's performance improves slowly over time, an artifact of cached result sharing effects as the cache fills up. As the other systems also benefit from this effect but are enhanced by predictive caching capabilities, they all outperform LRU.

6.3 Wikipedia Results

We measured query response times for the Wikipedia workload for each of the systems, results for which are shown in Figure 9c. Due to the Zipf($\rho = 1$) distribution of page-accesses and the extreme popularity of the GetPageAnonymous transaction (92% of transactions), LRU enjoys a cache hit ratio of 30%. Again, Apollo fails to recognize query patterns in the workload and shows only a small improvement in query response time over LRU because Apollo's predictive engine sequentially issues queries. By contrast, both ChronoCache and Scalpel-CC considerably improve the cache ratio to 50%, a consequence of their ability to recognize and exploit query patterns in the workload. Similarly, Scalpel-E maintains a cache hit ratio of 35%, but its inability to share cached query results across clients preclude it from reaching the high cache hit ratios of ChronoCache and Scalpel-CC. Note that the Wikipedia workload is an ideal case for the Scalpel-based systems since they are able to identify and exploit the key query patterns in the popular GetPageAnonymous transaction. Consequently, Scalpel-CC's performance is competitive with ChronoCache, both of which considerably outperform the other approaches by predictively caching more queries than the other approaches. This experiment demonstrates that ChronoCache's advanced modelling strategies have scant overhead on workloads without such patterns.

6.4 SEATS Results

The SEATS benchmark contains transactions with conditional access patterns — for example, a customer may access the system using their frequent flyer number, customer ID, or their login information. These conditional access patterns increase the complexity of learning the workload's access patterns. Despite these challenges, ChronoCache significantly reduces query response times compared to its competitors (Figure 10a). These gains are derived from cache hit rates: ChronoCache maintains an average cache hit ratio of 60%, while Scalpel-CC, Scalpel-E, LRU and Apollo maintain 45%, 40%, 35% and 35% cache hit rates respectively. ChronoCache significantly outperforms the other systems because of its support for per-loop constants, which appear in the FindFlights transaction. Although Scalpel-CC improves performance over LRU and Apollo, its lack of support for these types of query patterns hampers its performance. Shared caching semantics improve the performance



(a) SEATS Results **(b) AuctionMark Results** **(c) ChronoCache System Scalability**

Figure 10: Query response time results for ChronoCache, LRU, Scalpel-CC, Apollo, and Scalpel-E on the (a) SEATS Benchmark and (b) AuctionMark Benchmark. (c) depicts TPC-E query response times for 1 and 3 node ChronoCache deployments under increasing numbers of clients.

of Scalpel-CC compared to Scalpel-E, though frequent updates to the Flights table reduce these gains. Again, Apollo performs similarly to LRU due to its sequential query predictions and processing overheads.

6.5 AuctionMark Results

We measured query response times for each system on the AuctionMark workload over a 5 minute interval, averages for which are shown in Figure 10b. As expected, ChronoCache's support for per-loop constants substantially improves performance compared to its competitors. We observed that ChronoCache maintains a 45% cache hit ratio while Scalpel-CC and Scalpel-E maintain a 10% cache hit ratio — whereas both Apollo and LRU maintain less than a 2% cache hit ratio. The low cache hit rates of ChronoCache's competitors are due to queries being infrequently repeated in this workload. Furthermore, frequent updates to tables accessed in loops result in large numbers of cache misses due to invalidation when a loop is not cached in its entirety. These frequent updates also reduce the effectiveness of shared query caching, resulting in Scalpel-E performing comparably to Scalpel-CC.

6.6 Scalability Characteristics

We assessed ChronoCache's scaling capabilities by comparing a one-node ChronoCache deployment against a three-node deployment using the TPC-E workload, monitoring query response time as the number of clients increases (Figure 10c). The one-node deployment has lower latency for low numbers of clients because cached query results are not shared across Chronocache nodes and due to the three-node system's more restrictive session requirements (Section 5.2). As the number of clients increases, the three-node system significantly outperforms the one-node system by reducing resource contention. At 180 clients, the three-node deployment nearly halves the average query response time of the one-node deployment. ChronoCache scales well because its nodes do not need to interact with each other to process client requests.

6.7 Sensitivity Analysis

To test ChronoCache's sensitivity to its configuration parameters, we conducted TPC-E experiments with 10 clients in which we adjusted Chronocache's threshold for temporal correlations τ and the size of the cache. For τ, we observed that only extreme values ($\tau <= 0.01, \tau >= 0.95$) resulted in statistically significant variations in performance. For other values, average query response times were within 95% confidence intervals of the results presented in Section 6.1. Similarly, the cache size played little role in performance, unless it was set to be exceeding small (<10 MB). These results demonstrate that (i) ChronoCache does not excessively load query results into the cache to inflate its cache hit ratios, and (ii) ChronoCache is effective even when it is assigned only a small cache to manage. ChronoCache achieves this feat by loading query results into the cache *just before* they are needed, and is thus resilient to the size of the cache.

7 RELATED WORK

ChronoCache differentiates itself from prior approaches as a *fully online* solution that predictively caches query results using only information available at the middleware layer. We discuss these differences in more detail below.

Predictive Caching

Scalpel [9] is a client-side query prefetching system that exploits relationships among queries to optimize them via rewriting. Scalpel relies on a profiling mode to uncover these relationships, during which prefetching is disabled. Furthermore, it does not share prefetched query results across clients or transaction boundaries. Unlike ChronoCache, Scalpel does

not combine nested queries containing constants that vary per loop invocation nor fully support combining nested and batch queries [10]. ChronoCache overcomes these limitations by using adaptive online models which support a broader class of query relationships and are used for prediction immediately.

While Apollo [22] does not require a training period and functions as an online system, it submits predicted queries sequentially instead of combining them and does not support common client query patterns (i.e. loops). Consequently, Apollo does not perform comparably to ChronoCache (Section 6). Fido's predictive techniques [32] also do not combine queries, in addition to requiring an offline training period.

Ramachandra et al. use static analysis [11, 36, 37] to examine program code and modify it to retrieve query results using batching or asynchronous requests. In doing so, clients may avoid round trips to the remote database and experience reduced query response times. However, these approaches require offline analysis and modify client application code. As such, they are not applicable to middleware caching environments where client application code is unavailable, which ChronoCache targets.

Bilgin et al. [7] discuss how queries should be combined as part of a prefetching/read-ahead solution. In contrast to ChronoCache, their work assumes knowledge of an query transition graph and a data dependency graph.

Query prefetching can be viewed as an analogue of the materialized view-selection problem [13]. In contrast to ChronoCache, much of the work in this area requires a priori knowledge of the workload [4, 24] which is unavailable to a middleware application. Prior work without this limitation [17, 27] does not employ fine-grained predictive models like those within ChronoCache to anticipate future queries and cache their results.

Transaction Relationships

Houdini [35] uses Markov Models in a distributed database system to model stored procedures and optimize their execution by selecting the node at which it will execute, predictively locking partitions, disabling undo logging for transactions unlikely to abort, and speculatively committing transactions. These models are built offline from a workload trace, do not cross transaction boundaries, and model relationships among query input parameters and stored procedures arguments. Consequently, they do not capture relationships among query result sets and input parameters necessary for predictive caching.

Faleiro et al. [20] model data dependencies among transactions to support lazy transaction evaluation in deterministic databases. ChronoCache differs from this work in three key ways: (i) ChronoCache predictively executes queries rather than lazily deferring transaction execution, (ii) ChronoCache models relationships among queries rather than among database records in transactions, and (iii) ChronoCache does not require a *deterministic* database.

Middleware Caching

DBCache [30] and MT-Cache [29] are proprietary middleware caching systems developed for IBM DB2 and SQL Server, respectively. Watchman [39] proposes an intelligent cache admission scheme that decides which query results to admit and to replace in the cache. In contrast to least-recently-used (LRU) caching schemes, Watchman aims to minimize the cost of a cache miss rather than the number of cache misses. Watchman, DBCache, and MT-Cache do not predictively combine and prefetch queries into the cache.

Recent work has proposed extensions to caching systems to enhance their utility and performance. For example, ROBUS [28] fairly allocates cache space among clients in a multi-tenant system, and CPR [21] suggests means to compute range predicate queries over cached result sets. These orthogonal extensions could be deployed complementarily to ChronoCache's functionality.

Workload Modelling

Prior systems focused on workload modelling use machine learning techniques to predict query arrival rates [31], model query behaviour [38], and detect when the workload has changed [25]. In contrast to these systems, ChronoCache uses its workload model to discover related queries that will benefit from execution as a combined query, and subsequently, predictive caching.

8 CONCLUSION

Through ChronoCache, we presented techniques to dynamically (i) uncover query patterns in database application workloads, (ii) exploit query patterns by combining query requests, and (iii) predictively execute these combined queries and cache their result sets ahead of time. Such client application query requests are then satisfied from nearby edge nodes instead of via costly trips to a remote datacenter. ChronoCache enables clients to obtain these benefits without application modification. Performance results demonstrate ChronoCache's superiority over prior approaches on representative benchmark workloads.

ACKNOWLEDGMENTS

Funding for this project was provided in part by the Natural Sciences and Engineering Research Council of Canada, the AWS Cloud Credits for Research program, the Canada Foundation for Innovation, and the Province of Ontario.

REFERENCES

[1] 2019. http://memcached.org.

[2] 2019. https://github.com/antlr/grammars-v4.

[3] Database Test Suite 5. 2019. http://osdldbt.sourceforge.net/.

[4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505. http://dl.acm.org/citation.cfm?id=645926.671701

[5] Cristiana Amza, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Willy Zwaenepoel, Emmanuel Cecchet, and Julie Marguerite. 2002. Specification and Implementation of Dynamic Web Site Benchmarks. In *2002 IEEE International Workshop on Workload Characterization*. 3–13. https://doi.org/10.1109/WWC.2002.1226489

[6] ANSI. 1999. Information Systems Database Language SQL. *ISO/IEC 9075-1:1999* (September 1999).

[7] A. Soydan Bilgin, Rada Y. Chirkova, Timo J. Salo, and Munindar P. Singh. 2004. Deriving Efficient SQL Sequences via Read-Aheads. In *Data Warehousing and Knowledge Discovery*, Yahiko Kambayashi, Mukesh Mohania, and Wolfram Wöß (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–308. https://doi.org/10.1007/978-3-540-30076-2_30

[8] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (March 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[9] Ivan T. Bowman and Kenneth Salem. 2004. Optimization of Query Streams Using Semantic Prefetching. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/1007568.1007591

[10] Bowman, Ivan. 2005. Scalpel: Optimizing Query Streams Using Semantic Prefetching. http://hdl.handle.net/10012/1093

[11] Mahendra Chavan, Ravindra Guravannavar, Karthik Ramachandra, and S. Sudarshan. 2011. DBridge: A Program Rewrite Tool for Set-Oriented Query Execution. In *2011 IEEE 27th International Conference on Data Engineering*. 1284–1287. https://doi.org/10.1109/ICDE.2011.5767949

[12] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the New TPC-E Benchmark via an I/O Comparison Study. *SIGMOD Rec.* 39, 3 (February 2011), 5–10. https://doi.org/10.1145/1942776.1942778

[13] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. 2001. A Formal Perspective on the View Selection Problem. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 59–68. http://dl.acm.org/citation.cfm?id=645927.672216

[14] Marsha Collier. 2019. What an eBay End-of-Auction Email Tells You. https://www.dummies.com/business/online-business/ebay/what-an-ebay-end-of-auction-e-mail-tells-you/.

[15] Transaction Processing Performance Council. 2009. TPC-E On-Line Transaction Processing Benchmark. http://www.tpc.org/tpce/.

[16] Khuzaima Daudjee and Kenneth Salem. 2006. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 715–726. http://dl.acm.org/citation.cfm?id=1182635.1164189

[17] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shulka, and Jeffrey F. Naughton. 1998. Caching Multidimensional Queries Using Chunks. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*. ACM, New York, NY, USA, 259–270. https://doi.org/10.1145/276304.276328

[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] eBay. 2019. Customize Ebay's End of Auction Email. https://pages.ebay.com/CustomizedEOA/index.html.

[20] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2588555.2610529

[21] Shahram Ghandeharizadeh, Yazeed Alabdulkarim, and Hieu Nguyen. 2019. CPR: Client-Side Processing of Range Predicates. In *Cloud Computing – CLOUD 2019*, Dilma Da Silva, Qingyang Wang, and Liang-Jie Zhang (Eds.). Springer International Publishing, Cham, 340–354.

[22] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Scott Foggo, and Anil Pacaci. 2018. Apollo: Learning Query Correlations for Predictive Caching in Geo-Distributed Systems. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. 253–264. https://doi.org/10.5441/002/edbt.2018.23

[23] The PostgreSQL Global Development Group. 2019. PostgreSQL. https://www.postgresql.org/.

[24] Himanshu Gupta and Inderpal S. Mumick. 2005. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering* 17, 1 (January 2005), 24–43. https://doi.org/10.1109/TKDE.2005.16

[25] 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

[26] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[27] Yannis Kotidis and Nick Roussopoulos. 1999. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. ACM, New York, NY, USA, 371–382. https://doi.org/10.1145/304182.304215

[28] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. 2017. ROBUS: Fair Cache Allocation for Data-parallel Workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 219–234. https://doi.org/10.1145/3035918.3064018

[29] Per-Ake Larson, Jonathan Goldstein, and Jingren Zhou. 2004. MTCache: transparent mid-tier database caching in SQL server. In *Proceedings. 20th International Conference on Data Engineering*. 177–188. https://doi.org/10.1109/ICDE.2004.1319994

[30] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. 2002. Middle-tier Database Caching for e-Business. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. ACM, New York, NY, USA, 600–611. https://doi.org/10.1145/564691.564763

[31] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 631–645. https://doi.org/10.1145/3183713.

3196908

[32] 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.

[33] Terence Parr. 2019. https://www.antlr.org/.

[34] Andrew Pavlo. 2017. What Are Doing With Our Lives? No One Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM SIGMOD International Conference On Management of Data (SIGMOD '17)*. https://www.cs.cmu.edu/~pavlo/slides/pavlo-keynote-sigmod2017.pdf

[35] Andrew Pavlo, Evan P. C. Jones, and Stanley Zdonik. 2011. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *Proc. VLDB Endow.* 5, 2 (October 2011), 85–96. https://doi.org/10.14778/2078324.2078325

[36] Karthik Ramachandra, Mahendra Chavan, Ravindra Guravannavar, and S. Sudarshan. 2015. Program Transformations for Asynchronous and Batched Query Submission. *IEEE Transactions on Knowledge and Data Engineering* 27, 2 (February 2015), 531–544. https://doi.org/10.1109/TKDE.2014.2334302

[37] Karthik Ramachandra and S. Sudarshan. 2012. Holistic Optimization by Prefetching Query Results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2213836.2213852

[38] Carsten Sapia. 2000. PROMISE: Predicting Query Behavior to Enable Predictive Caching Strategies for OLAP Systems. In *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2000)*. Springer-Verlag, London, UK, UK, 224–233. http://dl.acm.org/citation.cfm?id=646109.679288

[39] 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

[40] Michael Stonebraker, Sam Madden, and Pradeep Dubey. 2013. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *SIGMOD Rec.* 42, 1 (May 2013), 44–49. https://doi.org/10.1145/2481528.2481537

[41] Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (SWAT 1971)*. 114–121. https://doi.org/10.1109/SWAT.1971.10