EagerDB:
A Predictive Cache Warming Tool
6.885 Final Paper

John J. Wang
wangjohn@mit.edu

December 11, 2013

Abstract

Many applications increase database performance using some form of caching. The majority of cache implementations, however, only increase performance for queries that have been already seen. EagerDB takes this a step farther and provides a predictive cache. It predicts queries that will occur in the future with high probability and automatically loads them into the cache. EagerDB allows the user to manually specify query preloads or automatically parse historical database logs, estimate a Markov model, and preload queries which have high transition probabilities.

1 Introduction

Most web applications improve database performance by caching – storing data has has previously been used. Using a cache can dramatically improve database performance when temporal locality is present (i.e. when previously requested data is requested multiple times). Caching has become one of the most fundamental concepts in computer science, and in turn, almost all large technology companies use caches in some form.

However, caches assume that historical data repeats. This assumption does not capture all of the potential performance gains that are possible. In fact, there is quite a large amount of data that exists which can be used to improve performance even further. Since there workflows of users tend to be methodical, the queries that get sent to a database can be predicted. One can then estimate the probability of some query X happening given the current state of the database, and "preload" X when it has a high probability of happening.

EagerDB provides a framework for preloading such queries. Developers can manually specify queries to preload or they can use a prediction engine for discovering queries to preload. EagerDB is built as a Ruby gem and can be easily included in a Ruby on Rails web application for easy improvements in database performance. This paper shall outline the design of EagerDB, the reasons for making particular design decisions, and the results of benchmark tests.

2 Background

There have been a number of papers which have had similar ideas.

3 Manual Query Syntax

EagerDB has its own easy to use syntax for manually specifying preloads. Behind this manual query syntax is the idea that any query Y can be used as an indicator for a later query X. Thus, if the probability of X occurring is high given that Y has occurred, then one should preload X by bringing it into the database's cache. We call Y the *match statement*, and X the *preload statement*.

3.1 Basic Definitions

In order to explain match and preload statements more, we need to make a distinction between non-binded and binded SQL queries. A *binded SQL query* is just any instance of a valid SQL statement which contains table, column, and attribute values. An example of a binded SQL query would be:

```
SELECT * FROM distributors WHERE name = 'walmart' AND
  town = 'Chicago' AND state = 'IL'
```

A *non-binded SQL query* is just an instance of a valid SQL statement which contains table and column values, but does not contain attribute values. For example, the non-binded version of the previous SQL query would be:

```
SELECT * FROM distributors WHERE name = ? AND
  town = ? AND state = ?
```

Non-binded SQL queries can be converted into binded SQL queries by providing *bind values*, an array of values which would be inserted into each bind location (denoted by a ? in the non-binded SQL query). To convert the previous non-binded SQL statement into a binded SQL statement, one would need to pass in the following array of bind values:

```
['walmart', 'IL']
```

3.2 Match Preload Files

Match and preload statements are non-binded SQL queries. Match statements allow EagerDB to match on a particular SQL statement structure, and preload statements allow us to preload a specific SQL statement structure when match statements occur. In this vein, one could manually specify match and preload statements. This is done by first specifying a single match statement, then any number of preload statements to load into the cache whenever that match statement is seen.

In a *match preload file* (MP file), a developer using EagerDB can specify a list of match statements and their corresponding preload statements. EagerDB can read over an MP file and incorporate any match and preload statements that have been made inside the file. A developer can use the MP file as a stand-alone product without running the Markov model so that the preload statements in the MP file will be the

only statements preloaded by EagerDB, or incorporate the MP file as an addition on top of the Markov model's statements. The syntax for an MP file is straightforward:

- Match statements are preceded by a dash "-".
- Preload statements are preceded by a rocket "=>". Preload statements will be paired with the last match statement seen in the MP file.
- SQL statements are encapsulated by quotation marks.
- Bind values, separated by commas, follow the SQL statement. The i^{th} bind value in the SQL statement corresponds to the i^{th} comma separated value after the SQL statement.

In addition to these syntax rules, there are methods provided for bind values which allow making general preload statements much easier. We provide the `match_result` and `match_bind_value` keywords.

The `match_result` keyword provides access to the result of the match statement of a preload. One can access the value of a column in the result by asking for the column name like so: `match_result.column_name`. For example, if a developer wanted to access the id of the result from his match statement, he could write a preload statement like so:

```
=> "SELECT * FROM things WHERE product_id = ?", match_result.id
```

The `match_bind_value` keyword is a similar construction, but provides access to the bind value of the binded instance of the match statement. One specifies an index i after the `match_bind_value` keyword, which gives access to the ith bind value from the match statement. For example, one could write:

```
- "SELECT * FROM products WHERE name = ?"
  => "SELECT * FROM things WHERE product_name = ?", match_bind_value(0)
```

In the above, whenever of a SQL query of the form `SELECT * FROM products WHERE name = ?` is seen, EagerDB will take the zeroth bind value (bind values are indexed by 0) from the statement and use it as the first bind value in the preload statement `SELECT * FROM things WHERE product_name = ?`.

```
- "SELECT * FROM users WHERE name = ?"
  => "SELECT * FROM products WHERE owner_id = ?",
       match_result.id

- "SELECT * FROM pinterest WHERE pin = ? AND interest = ?"
  => "SELECT * FROM tables WHERE pin = ? AND interest = ?",
       match_bind_value(0), match_bind_value(1)
  => "SELECT * FROM interests WHERE interest = ? AND
       pinterest_id = ?", match_bind_value(1), match_result.id
```

Figure 1: A snippet from an MP file

Figure 1 provides an example instance of an MP file. In this file, there are two match statements. The first match statement has a single preload, while the second

match statement has two preloads. EagerDB will listen to the stream of SQL queries coming from the database, and whenever a match statement matches the structure of a query, the preload statements associated with that match statement will be brought into the database's cache.

For example, using the MP file from figure 1, suppose the following sequence of queries arrived at the database (ordered chronologically):

```
SELECT * FROM products WHERE name = 'table' AND state = 'IL'
SELECT * FROM users WHERE name = 'john'
SELECT * FROM laptops WHERE BRAND = 'lenovo' AND serial_number = '5'
```

Then, after the second query (`SELECT * FROM users WHERE name = 'john'`) was run, its corresponding preload statement would be brought into the cache. For example, if the second query returned a result with an id of 52, then the following query would be brought into the database's cache: `SELECT * FROM products WHERE owner_id = 52`. The other queries in the sequence did not match the structure of any of the match statements, so EagerDB would not bring anything into the cache after their execution.

4 System Design

The EagerDB system is designed to be as unobstrusive as possible. It sits at the middleware level and only needs two things: 1) a connection to the stream of SQL queries and results being sent to and from the database and 2) a connection to the database for preloading queries. As a Ruby gem, EagerDB requires a minimal amount of integration and can be incorporated into a Ruby on Rails web application with less than 15 lines of code.

The system is also designed to be distributed and scalable. The EagerDB system is easily extended to multiple machines and can still be used with a sharded database. More details of the design later in the paper will show how this can be accomplished without any changes to the framework.

EagerDB is designed for flexibility, since it works on any database with a built-in cache, including MySQL, PostgreSQL, etc. It is SQL syntax agnostic, meaning that small variations in the syntax between different databases will not prevent EagerDB from preloading queries. This quality comes about because of the non-binded vs. binded SQL abstraction: all match statements are non-binded and general. In fact, EagerDB could potentially preload NoSQL queries, such as those from MongoDB, without any problems.

Additionally, EagerDB uses the database's own cache. This greatly simplfies setup and reduces the number of failure points. Instead of providing its own cache, which would have to deal with cache invalidation issues as well as duplication of data, EagerDB can rely on the pre-built mechanisms of the database. Thus, the design is resistant to bloating memory, and a user can tweak the size and properties of a single cache (instead of two). Moreover, using the database's own cache means that EagerDB can be used on more than just web applications. Since it has no appliction-specific logic, EagerDB can be used for anything that sends sends queries to a database.

Finally, EagerDB internally reads data from the MP file format, which enables a developer to change automatically generated preloads and provide their own preloads

relatively easily. Having a single format for representing data greatly simplifies the amount of work a developer needs to do to make changes to their own preloads.

In order to accomplish all of the above traits, EagerDB makes a number of abstractions which will be presented in this section. The codebase is written in Ruby (so that it can conveniently be used in Ruby on Rails web applications) and is broken up into two sections. The first is the processing code, which provides the infrastructure for parsing MP files and running EagerDB with a live stream of queries, and the second is the prediction code, which creates a Markov model and outputs an MP file based on historical database logs.

4.1 Processing Code

The processing code provides abstractions for listening to a stream of queries going to the database, encapsulating each query as a job to be processed, and finally processing those jobs and sending the resulting preload queries (if there are any) back to the database to be executed. Figure 2 provides an overview of the high-level architecture.

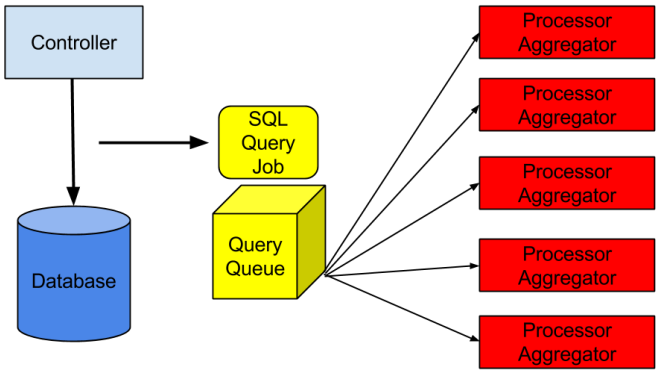

Figure 2: SqlQueryJobs are created from the incoming stream of queries, placed onto a queue, and finally sent to processor aggregators.

First, a `CommunicationChannel` object sits between the database and the controller and listens to queries that get sent to the database. Whenever a query appears in the `CommunicationChannel`, the query and its result will be processed and converted into a `SQLQueryJob`. This job is then placed on a query queue, which temporarily holds the `SQLQueryJobs`. A process runs and attempts to keep the query queue empty at all times, sending `SQLQueryJobs` as quickly as possible to an instance of a processor aggregator. Each processor aggregator will handle a `SQLQueryJob` and determine which preload statements (if any) need to be sent to the database.

Processor aggregators hold the many processors, the fundametal object for computation in EagerDB. Processors hold a single match statement and multiple preload statements. The processor is an encapsulation of a match statement and its preloads as defined in an MP file. There are two crucial methods for a processor in its public API, these are `matches?` and `process_preloads`. The `matches?` method takes in a

raw, binded SQL query from the `SQLQueryJob` (originally part of the stream of SQL queries going to the database) and checks if that query matches the processor's match statement. The `process_preloads` method will take in the same raw, binded SQL query, and will return binded preloaded statements if `matches?` returns true, and an empty list otherwise.

Each processor aggregator holds all of the processors from an MP file, and all processor aggregators are equivalent. When used with a single machine, EagerDB would only create a single processor aggregator, but one could create many more processor aggregators to scale. Figure 3 shows an overview of the processor aggregator. Basically, the aggregator parses the raw SQL from the `SQLQueryJob`, then hands off the resulting job to the correct processor. If the parsed SQL matches a processor's match statement, then that processor's preload statements get added to the preload query queue, and will eventually reach and be executed by the database.

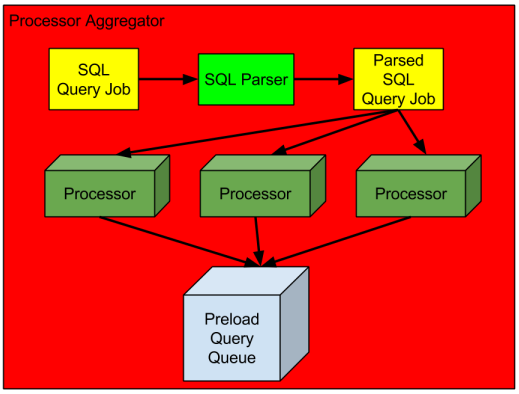

Figure 3: Processor aggregators process the SqlQueryJob (which contains raw, non-binded SQL from the stream) and sends parsed result to individual processors. Matches result in preloads getting sent to a preload queue.

The default implementation of the query processor uses a hash table so that statements can be matched very quickly. When a new `SQLQueryJob` comes in, the non-binded SQL is used as a key in the hash table. The hash table values for a particular non-binded SQL key is an array of all processors that match on that SQL statement. Thus, the total time required to process any statement and load preloads is $O(k)$ where k is the number of matching statements (assuming that the number of preloads per match statement is less than some constant). Most applications tend to have incredibly small k's, which makes this process extremely fast.

4.2 Sharding Databases and Scaling EagerDB

This subsection will spend some time discussing implications of design decisions on scaling. First, notice that increasing throughput for EagerDB jobs is incredibly easy: just add more processor aggregators. We have a couple nice properties that allow this to occur. First, each `SQLQueryJob` only needs to go to a single processor, since each aggregator is identical. Second, `SQLQueryJob`s can be dropped by the network without fear of losing real information (only possible performance improvements). Of course,

if the probability of a network failure is p, and the developer wanted an arrival rate of a, then one could send $\lceil a/p \rceil$ `SQLQueryJob`s over the network to obtain the required arrival rate.

Sharding the underlying database is also quite easy, as long as an appropriate function for mapping a SQL query to a particular shard exists. Each database shard would get its own query queue. The query queue would have access to all processor aggregators, and the processor aggregators would have references to all database shards. The preload query queue on each processor aggregator would send the preload query to the correct shard by invoking the function for mapping queries to shards. Thus, sharding databases, though not implemented in the current version of EagerDB, can be performed without major modifications to the underlying framework.

4.3 Prediction Code

This section will discuss how automatic prediction of high probability preloads are made. The general idea is to build a Markov model based on historical database logs, then find match-preload pairs with high transition probabilities. We think of the state of the Markov model at time t as the last raw, binded SQL query that was run before time t. Thus, we want to find an estimate of $P(X|Y)$ for all X in the set of possible queries and where Y the current state. We can then create a match pair, which we will denote as (Y, X) and have Y as the match statement and X is the preload statement, if $P(X|Y) > \lambda$ for some parameter λ.

However, the problem as it stands is close to intractable since the set of all possible X is infinite. We can make the problem easier however. Instead of thinking of X and Y as arbitrary, binded SQL statements, we can change the definition so that X and Y are non-binded statements. Then, once we have found match pairs (Y, X), we will go back and infer the bind values for X. Thus, we want to find $P(X|Y)$ where X, Y are non-binded SQL statements.

To estimate this probability, we will use historical database logs and a time parameter T. Here, T will be a time after which queries are no longer considered pairs. For example if Y occurs, then X occurs 10 years later, there is a very little chance that Y is actually predictive of X. Thus, we will only consider (Y, X) pairs where X occurs within time T after Y has occurred. The complete algorithm for estimating probability is as follows:

- Break apart and group database logs by user id or ip-address (or a similar metric, depending on which is available in the data). Each group should contain the queries made by a single user or ip-address.
- For each query Y in the group, find the other queries X which occur within time T of Y occurring, and increment the count, $C((Y, X))$, of the (Y, X) pair. Also, increment the count, $C(Y)$, of the number of times Y occurs.
- After processing each group, and all queries in each group, you have the total occurrences of each (Y, X) pair, and the total occurrences of each query Y. Now use the estimate $\hat{P}(X|Y) = C((Y, X))/C(Y)$. Check if $\hat{P}(X|Y) > \lambda$ for any Y, X pair, and return all pairs as a (Y, X) match.

Notice that this is easily parallelizable in the MapReduce framework. The queries for a particular user or ip-address are passed as documents in the map step. Then, for

each query Y, we can emit the Y as the key, and an array of all the X's which occur within T time of Y occuring as the value. In the reduce step, we compute $\hat{P}(X|Y)$ for all possible X's that have been sent to this particular node, since we also can record a count of the total number of times Y occurred. In this way, massive database logs can be broken up and used for finding potential match pairs.

5 Conclusion

Future directions of research include adding metadata to queries, figuring out match result issues with schemas, optimizing results.