Query Optimization by Analyzing and Pre-caching Frequent Queries in Compile-Time

Ahmad Shahab Tajik
University of Michigan
tajik@umich.edu

Yike Liu
University of Michigan
yikeliu@umich.edu

HyunJong (Joseph) Lee
University of Michigan
hyunjong@umich.edu

ABSTRACT

Applications and users are mostly oblivious to what tables are frequently utilized and queried in the Database Management System (DMBS). As the era of big-data where hundreds of big tables are queried, utilized, and fetched from high-latency storage, precaching the data with high-likelihood of usage is crucial to improve application performance. However, without application or user specific domain knowledge in frequently/likely used data/tables, designing a generic table caching system is challenging.

In this report, we show that with a dynamic analysis of queries at compile-time, it is possible to identify a set of influential queries such that if we pre-cache the data of those queries, the application will become considerably faster.

We designed, implemented, and evaluated the Speculative Database Table Precaching (SDTP) framework that takes a generic approach in identifying those influential queries for the input application, and producing an optimized version of that application that does the same job, except it also pre-caches the data used by those queries to mitigate traditional high-latency bottleneck coming from non-volatile storage. Our evaluations on two real-world datasets show that our approach improves the application performance between 4-22%, without extra support from hardware.

1. INTRODUCTION

In Database Management System (DMBS), users are often unaware of what tables are frequently utilized and queried. Without knowing underlying tables that are used, identifying the bottleneck of querying the database(s) is challenging. This becomes worse as the era of big-data where hundreds of bigtables are queried concurrently [2].

In this report, we argue for a speculative database pre-caching (SDP) system that removes a traditional bottleneck in database system: fetching from storage. We propose that by speculatively pre-caching the data used by "more influential" queries ahead of time, an application improves performance with an illusion of asynchronous operation and query processing.

Furthermore, we suspect that applications in similar category are likely to utilize same data. This means that by pre-fetching and caching the data that is shared among multiple applications, we remove the bottleneck in traditional database systems at compiler optimization domain.

2. RELATED WORKS

Our work on optimization of query applications is most related to two research topics: query optimization and pre-caching techniques.

2.1 Query Optimization

Query optimization has been a classic topic in the research field of database. Many optimization work only focuses on both specific languages and types of databases, e.g. [7] tries to speedup processing of the SPARQL query language for RDF, [12] looks into effective querying and mining methods over such large-scale graph-structured data, and [4] studies federated data management and query optimization for linked open data. Different from these work, our optimization is code-based, and theoretically capable of handling all types of databases and queries.

On the other hand, our optimization assumes unlimited buffer size for the application to exploit. This might not very efficient on extremely large dataset or specific setting where the buffer size is constrained. With this reason, people have turned to other techniques. E.g., [8] propose a query optimization scheme for MapReduce-based processing systems by exploiting parallelism. And to solve the multi-query optimization for SPARQL, [5] proposes heuristic algorithms that partition the input batch of queries into groups such that each group of queries can be optimized together.

2.2 Pre-Caching

Existing works in domain of pre-caching have been done for computation heavy tasks in order to either reduce latency for user- interacting applications [6, 1] or reuse existing output from previous tasks [11, 10].

Recent system that pre-caches for GPU rendering [6] speculates next frames based on current context in user frames. Our implementation takes a similar approach, where we apply speculation on tables and queries to pre- cache. Recent work in virtual reality pre-caching [1], which substitutes weak computation-power in mobile devices by plentiful large storage with pre-caching most of possible objects at runtime, is similar to our system; however, our system analyzes "cacheable" queries at compile- time.

Idea of reusing outputs from previous tasks have been widely explored in big-data analysis [11, 10]. Recent work in distributed system [11] caches outputs from prior computation to deliver an abstraction of pre-caching to next jobs, whereas our system exploits the computational heavy jobs to pre-cache next set of queries from tables. Resilient Distributed Datasets [10] adapts memoization of recent outputs in abstract datatype to reduce latency for future, similar tasks. In comparison, our system mainly focuses on results of executing a set of queries to achieve the same goal.

3. OPTIMIZATION PASS DESIGN

Our compiler pass performs a dynamic profiling by running the application and monitoring the SQL queries that are sent to the database, and collecting the queries, the number of their usage and their start and end times.

Using the collected information, it generates a schedule for pre-fetching a subset of queries. Then, the pre-fetching schedule among with the pre-fetcher library is injected into the application's code. The resulting object files will be the original application that when it runs, in addition to doing the job it was doing, it also tries to pre-fetch the queries according to the pre-fetching schedule.

The pre-fetching process take place in parallel to the applications main function (more specifically, in a separate background thread). Pre-fetcher tries to not interfere the actual DB communication of the application. It suspends running the background pre-fetching queries when the application submits a query itself, to avoid causing latency to the actual query.

Furthermore, after submitting an actual query by the application, pre-fetcher removes that query from it's schedule, because pre-fetching it doesn't make sense any more (the scheduling algorithm - as described bellow - tries to generate a schedule that avoids this circumstance as much as possible, but due to the random factors and unpredictable nature of the applications, this is always possible to happen).

3.1 Pre-Fetching Scheduler

By a pre-fetching schedule, we mean a sub-set of the queries that the application executes with a specified order. This schedule is made by the compiler pass using the information it gains about the queries in the dynamic profiling. The schedule will be given to our pre-fetcher library that will be injected in the application's code. The pre-fetcher library will execute the queries in the schedule, in the given order.

We try to put the most queries possible in the schedule, but with respect to the following constraints:

1. The pre-fetcher database queries will have a lower priority that the "actual" queries made by the application in the run-time. So, we make sure that we do not put more queries in the schedule than can be run in the time that the application is not running DB query itself. We do this by calculating the approximate time gap between the subsequent "actual" query calls, and calculating the estimated ratio of the time that the application runs DB queries over the total runtime.

2. Each query should be included in the schedule only if there is a good probability that it get a chance to be pre-fetched before the application runs that query itself. Because after the application runs the query there is no point in pre-fetching it. Therefore, we only put a query in the schedule if the estimated time of prefetching it is less than the estimated time that the actual query is run.

3. For any two queries q_1 and q_2 in the pre-fetching schedule, if q_1 is before q_2, then it should be the case that q_1 is also executed before q_2 in the application. This constraint helps the pre-fetch queries be closer to their actual queries, so that there is a smaller chance that the cached data is over-written by the between queries, and the actual query can benefit more from the cached queries.

4. IMPLEMENTATION

We implemented our technique as a Java library and compiler. Current implementation is tested with MySQL database, however it is DBMS-agnostic and we expect it to work with any underlying DBMS that supports JDBC drivers and data caching.

Our implementation has two parts:

1. The compiler pass, including the Query Profiler and Pre-fetching Scheduler

2. The Query Pre-fetcher library that is injected in the application and pre-fetches the queries based on the given schedule

The compiler pass is written in Java and Shell script. It uses the regular Java Compiler (`javac` command) to compile the application. The profiler does not need to process the source-code or basic blocks because it can monitor the application and DBMS communications using JDBC drivers. Injecting the pre-fetcher is done by including it to the application's code with making some changes in the .java files headers, and putting the generated schedule in a specific place in the pre-fetcher source code.

The query pre-fetcher library is a Java library that uses Java multi-threading to run the pre-fetching queries in the background while the application is doing its normal functions.

5. EVALUATION

We evaluate our query optimization pass on two datasets: Github [3] and Yelp [9]. Yelp dataset contains 5 tables: business, review, user, check-in, and tip, where each table has its own attributes as described in Table 1. Github sample dataset contains more than 20 tables (details in [3]), but for evaluation purpose, we selectively used following tables: CreateEvent, DeleteEvent, PushEvent, ForkEvent, and PullRequestEvent. Due to the nature of "git", each event has an unique attribute followed by "repository" table. Attributes of each table are enumerated in Table 2.

The applications on the datasets contains multiple queries of different types on either individual tables or joined tables. Between each query we insert some "sleep" time (or *non-DB time* as we call in this report), which is the simulation for the time that the application does not interact with DB and is doing something else, e.g. waiting for some user's next response/input, doing some process on the output of the previous DB query, waiting for some network/IO operation to be done, etc.

Table	Attributes	Size
business	{type, business_id, name, neighborhoods, full_address, city, state, latitude, longitude, stars, review_count, categories, open, hours, attributes}	229222
review	{type, business_id, user_id, stars, text, date, votes}	9028311
user	{type, user_id, name, review_count, average_stars, votes, friends, elite, yelping_since, compliments, fans}	686557
check-in	{type, business_id, checkin_info}	61050
tip	{type, text, business_id, user_id, date, likes}	667238

Table 1: Yelp dataset

Table	Attributes	Size
CreateEvent	{master_branch, pusher_type, repository, private, fork, url, created_at, size, language}	medium
DeleteEvent	{pusher_type, repository, private, fork, url, created_at, size, language}	medium
5PushEvent	{before, after, created, deleted, forced, compare, commits, head_commit, repository, pusher, sender}	large
ForkEvent	{forkee, owner, full_name, url, private, fork, repository, private, fork, created_at }	medium
PullRequestEvent	{action, pull_request, body, created_at, updated_at, closed_at, merged_at, head_repo, base_repo, commits, repository, private, fork, url, size, sender}	large
repository	{id, name, full_name, owner, login, id, avatar_url, gravatar_id, url, html_url, followers_url, following_url, gists_url, starred_url, subscriptions_url, organizations_url, repos_url, events_url, received_events_url, type, site_admin, private, html_url, description, fork, url, forks_url, keys_url, collaborators_url, teams_url, hooks_url, issue_events_url, events_url, assignees_url, branches_url, tags_url, blobs_url, git_tags_url, git_refs_url, trees_url, statuses_url, languages_url, stargazers_url, contributors_url, subscribers_url, subscription_url, commits_url, git_commits_url, comments_url, issue_comment_url, contents_url, compare_url, merges_url, archive_url, downloads_url, issues_url, pulls_url, milestones_url, notifications_url, labels_url, releases_url, created_at, updated_at, pushed_at, git_url, ssh_url, clone_url, svn_url, homepage, size, stargazers_count, watchers_count, language, has_issues, has_downloads, has_wiki, has_pages, forks_count, mirror_url, open_issues_count, forks, open_issues, watchers, default_branch }	large

Table 2: Github dataset

For each experiment we ran the tests three times and averaged all the results to reduce the effect of random factors and make the evaluation more reliable. In each run we track the runtime of our application both before/after optimization.

We ran our experiments in a server with 64 CPU cores and 1TB of RAM and SSD storage. The database that was used in the experiments is MySQL (version 10.1.19-MariaDB).

In the following sub-sections we first present the evaluation results for the speedup we could gain by applying our technique to our benchmark applications. We also studied the effect of non-DB time and database's cache buffer size on the speedup our method can achieve. The result for those studies are presented in subsections 5.2 and 5.3.

5.1 Speedup Achieved

As shown in Figure 1, for the Yelp application, we could achieve significant speedup (saved 16-22% time). The experiment is done for different sizes of dataset to show that our approach works for different ranges of data size. Also, as shown in Figure 2, our approach brings about a fare amount of speedup (4-7.5%) for the Github application as well.

One explanation for the smaller amount of improvement in the Github application, compared to the Yelp application, can be the fact that the queries in the Github application have less overlap in terms of the data they use, so pre-fetching a sub-set of queries does not benefit the other queries much.

In the experiments reported in this subsection, non-DB time was 50% and buffer size was 200% for the Yelp dataset and 100% for the Github application. As we'll see in the following sub-sections, these parameters have effects on the speedup we can gain.

It's worth mentioning that the evaluations are done in a server with SSD disks, which have a very high speed, compared to more commonly used hard disks. Therefore, we expect to observe much higher speedup when our technique is evaluated in an environment that uses hard disks rather than SSD.

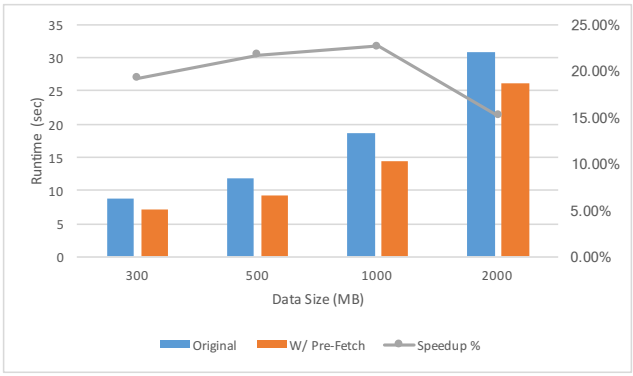

Figure 1: Speedup on Yelp dataset

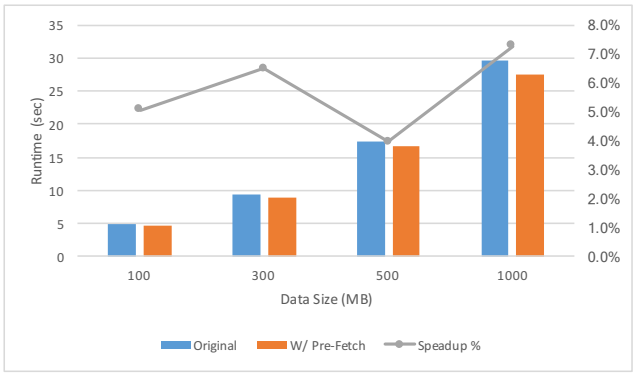

Figure 2: Speedup on Github dataset

5.2 Effect of Non-DB Time on Speedup

As mentioned in the section 3.1, the amount of non-DB time is important for our approach because the pre-fetching happens only in these times (where the application is not running queries itself), and scheduler considers this parameter when it generates the pre-fetching schedule.

In order to be able to evaluate our approach in applications with different non-DB time ratio, we made the amount of "sleep" time between queries, arguments of the test applications so they can be changed easily.

We evaluated the effect of non-DB time on the speedup we can gain by our technique for the Yelp application. The results are shown in Figure 3. We examined different sizes of datasets and observed the same trend. We see that for all size of database tested, we always have the runtime improvement increases initially as we increase percentage of non-DB over the total runtime, reaches a maximum saving of runtime, and then decreases as we further increase non-DB time.

The explanation for this trend is that when non-DB time percentage is small, the pre-fetcher only could pre-fetch a small amount of all queries. So, most of the queries won't benefit from pre-fetching. On the other hand, when the non-DB time percentage is too big, it means that the database queries cause a small amount of application's total runtime, therefore even by making them much faster we would not be able to have a big impact on the total runtime.

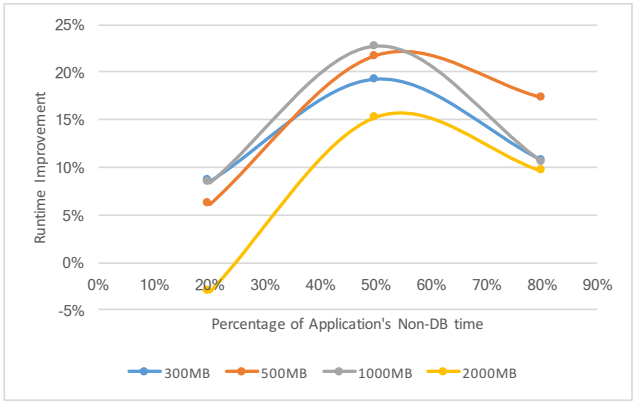

Figure 3: Effect of non-DB time on Speedup on Yelp dataset

5.3 Effect of DB Cache Buffer Size on Speedup

In this section we present the results of our evaluation on the effect of the size of buffer cache on the speedup we can gain. (Figure 4)

We repeated the experiment with different sizes of data to be more reliable. Because the data size is not fixed, buffer size is expressed as the percentage of the data size.

The trend is expected because when we have a small buffer, pre-caching the data might not be useful to the actual query, because by the time that the actual query needs the data it might already be flushed out from the buffer due to the need of in-between queries. When we have a big enough cache buffer, our method can provide a good speedup. However, after that point, increasing the buffer size won't help.

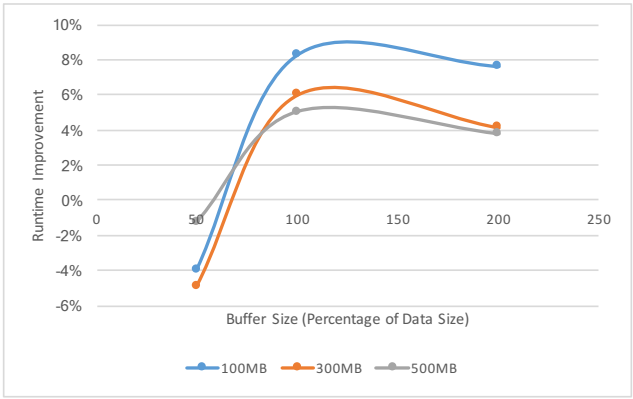

Figure 4: Effect of DB cache buffer size on Speedup on Github dataset

6. CONCLUSION

In this report, we proposed a general, effective, compile-time technique for reducing database applications' runtime. Our technique involves a dynamic profiling to monitor and gather information about the SQL queries in the application, and then using those information to generate a schedule for queries to be pre-fetched in the application in order to make it faster.

The evaluations on two real-world datasets (Yelp and Github) shown that our approach can improve the application performance considerably (up to 22%), even when tested with SSD. Although not tested, it is logical to think that applications will gain even more speedup when our technique is applied in an environment with hard disks rather than SSD's. Because hard disk is much slower than SSD and pre-caching the data into memory can have a bigger impact on the performance when using hard disks.

7. REFERENCES

[1] K. Boos, D. Chu, and E. Cuervo. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 291–304, New York, NY, USA, 2016. ACM.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[3] Github. Github Event Types & Payloads (https://www.githubarchive.org), 2016.

[4] O. Görlitz and S. Staab. Federated data management and query optimization for linked open data. In *New Directions in Web Data Management 1*, pages 109–137. Springer, 2011.

[5] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for sparql. In *2012 IEEE 28th International Conference on Data Engineering*, pages 666–677. IEEE, 2012.

[6] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165. ACM, 2015.

[7] M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, pages 4–33. ACM, 2010.

[8] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.

[9] Yelp. Yelp's Academic Dataset (https://www.yelp.com/academic_dataset), Mar. 2013.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[12] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351, 2010.