Time-, Energy-, and Monetary Cost-Aware Cache[*] Design for a Mobile-Cloud Database System

Mikael Perrin[1], Jonathan Mullen[1], Florian Helff[1], Le Gruenwald[1], Laurent d'Orazio[2]

[1] School of Computer Science, University of Oklahoma, Norman, Oklahoma, USA
{mikael.perrin, jonathan, fhelff, ggruenwald}@ou.edu
[2] CNRS, UMR 6158, LIMOS, Blaise Pascal University Clermont-Ferrand, France
laurent.dorazio@isima.fr

Abstract. Growing demand for mobile access to data is only outpaced by the growth of large and complex data, accentuating the constrained nature of mobile devices. The availability and scalability of cloud resources along with techniques for caching and distributed computation can be used to address these problems, but bring up new optimization challenges, often with competing concerns. A user wants a quick response using minimal device resources, while a cloud provider must weigh response time with the monetary cost of query execution. To address these issues we present a three-tier mobile cloud database model and a decisional semantic caching algorithm that formalizes the query planning and execution between mobile users, data owners and cloud providers, allowing stake holders to impose constraints on time, money and energy consumption while understanding the possible tradeoffs between them. We use a hospital application as a user case.

Keywords: mobile, cloud, energy consumption, monetary cost, cache.

1 Introduction

The main principle of mobile computing is the possibility to provide to the user resources independent of his/her location. A mobile device has the advantage of being light and easily transportable, but has constraints on its limited resources, such as memory, computing power, screen size, network connection and energy. Those constraints are even more important when they are compared to the resources available on a server, and even more with a computation service on the cloud with elastic resources [1].When the mobile device's lack of resources becomes critical, using cloud services becomes necessary to answer to the user constraints.

On the cloud, the different pricing models used by the cloud service providers bring a new challenge for the management of the resources, which is to solve a two-dimensional optimization problem concerning two constraints: query execution time and monetary cost that the cloud service tenants must pay to the cloud service providers. On the mobile device, one of the most important resources for its users is the remaining battery life. The remaining energy can be used either to evaluate queries or to check the

[*] This work is partially supported by the National Science Foundation Award No. 1349285

results. Therefore, this adds up a third dimension to the optimization problem: the mobile device’s energy consumption. Also, in order to reduce network communication between the user and the cloud, which consequently reduces the query processing time, we can use a cache system [2] called semantic caching. A cache system allows previously processed data to be stored. Those data can be computed or downloaded from a server. If they are to be requested again, the cache will return them in a transparent way in order to minimize the query processing time. At least two different event types can occur in the cache. When a looked up item is found within the cache, it is called a cache hit; otherwise, it is a cache miss. A semantic cache allows the cache to be used even though some of the data are not available in the cache. Therefore, the input query will be divided into two sub-queries: one to retrieve data from the cache and one to be sent to the server to retrieve the missing data. Therefore, using semantic caching allows the size of the data transferred between the cloud and the mobile device to be reduced.

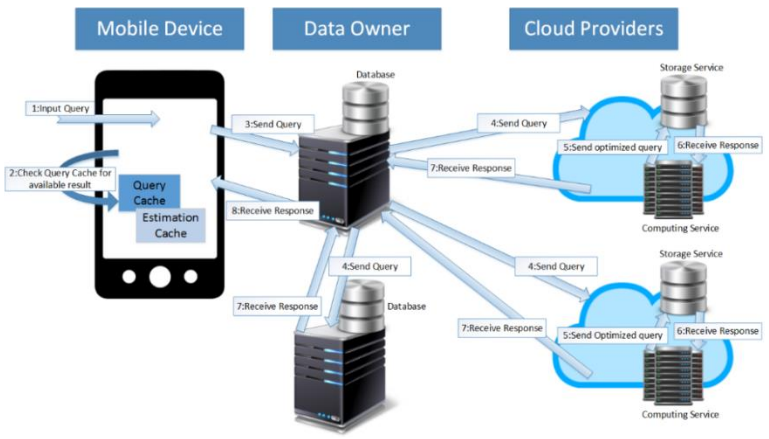


Fig. 1. 3-tier mobile cloud database architecture

To solve this optimization problem in an environment where data locality matters, we propose a 3-tier architecture (Fig. 1) and apply it to a user case of medical applications. Indeed, we can imagine that during a meeting, a hospital director wants to access some doctor information items and projects. To do this, he/she uses a tablet (**mobile device)** and builds the query with the user interface provided by the application installed on it. The hospital owns some private data, such as the doctor's identity, and stores it on a server infrastructure with physical limitations. This static infrastructure is called the **data owner**. Some of the information is not considered private, such as this meeting’s date and time with the director, and can be stored on elastic infrastructures on the cloud called **cloud providers**. Those services can answer to the real need in computing resources or storage resources for the hospital. The data owner can access those services.

With the proposed architecture, our goal is to solve the following research question: How can we process fast queries from a mobile device while respecting the user constraints on query execution time, energy consumption and monetary cost? More particularly, how can we use the cache system to estimate whether it is more profitable to process the query on the mobile device or on the data owner/cloud servers? The key contribution of this work is the development of a second cache on the mobile device called estimation cache working with the semantic query cache. The rest of the paper is organized as follows: Section 2 discusses the background and related work; Section 3 presents the proposed algorithm, called MOCCAD-Cache; Section 4 presents the experimental results evaluating the performance of the proposed solution; and finally, Section 5 concludes the paper and suggests future work.

2 Background and Related Work

[3] and [4] offered a formal definition of semantic caching which consists of three key ideas. Firstly, semantic caching contains a semantic description of the data contained in each cache entry such as the query relation, the projection attributes and the query predicates. This semantic description is used to designate the data items contained in the corresponding entry and to know quickly if the entry can be used to answer to the input query. Secondly, a value function is used for cache data replacement. For example, if the chosen value function is time, we can choose to replace the oldest entries. Finally, each cache entry is called a semantic region and is considered as the unit for replacement within the cache. It stores the information concerning the results of a query: a list of projection attributes, a list of query constraints (predicates), the number of tuples for this region, the maximum size of a tuple, the result tuples and additional data used to handle data replacement within the cache. When a query is processed, the semantic cache manager analyses the cache content and creates two sub-queries: a *probe query* to retrieve data in the cache; and a *remainder query* to retrieve the missing data from the database server. [4] and [5] define the notion of *query trimming* which corresponds to the cache analysis, necessary to split the input query into a probe query and a remainder query. More recently, other contributions have been added to this area [6], [7] and [8]. However, even though their contributions improve the performance of the query trimming algorithm, the version in [4] provides much more accuracy towards its implementation. During this query trimming algorithm, four types of event can occur: cache exact hit and cache miss as explained previously, as well as cache extended hit and cache partial hit added by the semantic caching definitions.

Cache Extended Hit. There are several types of cache extended hit: the result of the input query is included inside one of the regions in the cache (Fig. 2.a); the result of the input query needs to be retrieved from several regions in the cache (Fig. 2.b); and the input query and the semantic region's query are equivalent (Fig. 2.c). Thus, if we retrieve the semantic region's query result, we also retrieve the input query result.

Cache Partial Hit. Part of the query result can be retrieved from the cache with a probe query, but another part needs to be retrieved from the database server with the remainder query. For example, with the input query as $\sigma_{HR\geq 57}(NOTE)$ and our example's cache content shown in Fig. 3, query (3) can be used to retrieve only a part of the query result. Therefore, all the tuples corresponding to the remainder query $\sigma_{HR>57}(NOTE)$ would be downloaded from the database server.

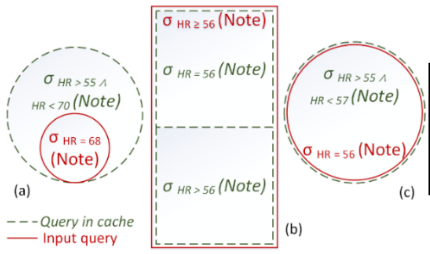


Fig. 2. Example: Cache Extended Hit

$$\sigma_{HR<57}(NOTE) \tag{1}$$

$$\sigma_{age>28 \wedge age<32}(DOCTOR) \tag{2}$$

$$\sigma_{HR=57}(NOTE) \tag{3}$$

Fig. 3. Example: Cache Content for Partial Hit

In order to demonstrate the performance of semantic caching for a server in ideal conditions, [3] and [9] compared this type of caching with page caching and tuple caching. For page caching, when a query is posed at the client side, if one of the pages (statically sized transfer unit) has not been found, then all the data corresponding to the page will be obtained from the server. For tuple caching, the unit is the tuple itself. This type of caching has good flexibility but a huge overhead in terms of performance. More recently, [10] showed the performance of semantic caching for types of applications requiring important workloads and using a limited bandwidth environment. Their results support our choice of semantic caching inside our Mobile-Cloud environment as a good way to reduce the overhead in query processing time. [11] and [12] studied semantic caching performance in a mobile environment and mainly focuses on location dependent applications to emphasize how useful semantic caching can be in such an environment.

Semantic caching, however, can create some overhead in several cases. This overhead has not been considered, mainly due to the assumption made in the previous works that it is always more efficient to process a query on the mobile device's cache rather than on the cloud. For many years, the bandwidth and the resources provided by the server disallowed one to even think about using the server to process the query when the result is already close to the user. However the power of cloud computing changes the game's rules. Several estimations now need to be computed in order to determine where the query should be processed in order to meet the user's constraints.

3 Proposed Solution

This section describes the processing time, energy consumption and monetary cost estimation computations as well as the proposed caching algorithm, MOCCAD-Cache. We assume that only selection queries are posed without any join operations. We also assume that no network disconnection occurs and that the database on the cloud is not altered. From the mobile device, only the data owner can be accessed. However, since we do not assume any difference between the data owner and the cloud servers while focusing on caching, the data owner and the cloud servers will be considered as one in our algorithm.

3.1 Estimation Computation

To compute the time and energy needed to process a query on the cloud, it is necessary to consider the time and energy needed to transfer the data between the mobile device and the cloud. To do this, we need to estimate the size of the query result sent back from the cloud. Then, knowing the bandwidth, it is possible to determine the time to download the data. [13] provided a formula to compute this size. The total time to retrieve the data from the cloud can be computed by adding the time to send the query to the cloud (negligible due to the small amount of data transferred), the time to process the query on the cloud, and the time to download the result. The estimated time and money to process a given query on the cloud are posed to the cloud services.

From the query execution time, it is possible to estimate the corresponding energy consumption. Three different steps define query processing time on the mobile device. $t_{compute\_estimation}$ is the time to compute the estimation (i.e. time to process the cache to determine the probe and remainder queries and compute their respective estimations). $t_{process\_mobile}$ is the time to process a query on the mobile device. $t_{process\_cloud}$ is the time to process a query on the cloud. Also, different steps described in [14] and [15] allow us to model the power consumption during the different activities and suspended states on the mobile device. Here we use the electric intensity I in Amperes. I_{idle} is the energy consumed when the mobile device is awake but no application is active (considered negligible). I_{CPU} is the energy consumed when the mobile device processes a query on its cache. $I_{network\_low}$ is the energy consumed when the mobile device waits while the query is processed on the cloud. $I_{network\_high}$ is the energy consumed when the mobile device retrieves the query results from the cloud. In order to know the amount of battery has been consumed, we need to determine the intensity of each component used in the previously defined states. The consumed energy amount allows us to quantify the amount of energy that we drain from the battery Q in Ampere Hours, which can be computed by multiplying I the current in Amps and t the processing time in hours. Therefore, we can simplify the computation of the energy consumption estimation by using only the electrical charge.

$$Q_{compute\_estimation} = I_{CPU} \times t_{compute\_estimation} \tag{1}$$
$$Q_{process\_mobile} = I_{CPU} \times t_{process\_mobile} \tag{2}$$
$$Q_{process\_cloud} = I_{network\_low} \times t_{exec\_query} + I_{network\_high} \times t_{send\_result} \tag{3}$$
$$Q_{total} = Q_{compute\_estimation} + Q_{process\_mobile} + Q_{process\_cloud} \tag{4}$$

These time and the energy consumption estimations depend mainly on the type of event that occurred during query trimming. If a cache hit occurs, we consider that retrieving the query result is negligible in terms of time and energy consumption. We consider also negligible the processing cost in the equivalent extended hit case (Fig. 2.c). The cases where a semantic region has to be processed (i.e. included extended hit or partial hit) can be a little bit more complex. The complexity of this type of evaluation depends mainly on the data structure used to store the tuples within the cache [16], as well as on the presence or not of indexes for the attributes that belong to the input query predicates [13]. In our proposed algorithm, the tuples are stored randomly in a contiguous list. It will therefore be necessary to check serially each tuple to see if it satisfies the input query predicate. The time corresponding to those operations can be determined statistically by processing queries on several segments of different sizes before any evaluation.

3.2 Cache Managers

The organization of managers presented in [17] is used for cache management which consists of three independent parts. The Cache Content Manager is used to manage the results inside the cache. It is used for insertion, deletion and look up. The Cache Replacement Manager is used to handle cache data replacement by some more relevant items. The Cache Resolution Manager is used to retrieve the data in the cache as well as to invoke some external tool to load the missing data items, for example.

Also, we introduce three other types of managers to handle the computation of every estimation as well as the elaboration of a better solution. The Mobile Estimation Computation Manager is used to compute the execution time and the energy consumption while executing a query on the mobile device. The Cloud Estimation Computation Manager is used to compute the time, the mobile energy consumption and the monetary cost necessary to process a query on the cloud. Finally, the Optimization Manager takes care of the choice between the mobile query processing estimation and the cloud query processing estimation while respecting the user constraints, such as the remaining battery percentage, the maximum query response time and the money to be paid to the cloud service providers for the query evaluation. Three cache structures, query cache, mobile estimation cache, and cloud estimation cache described below, are used to provide a solution for the given problem. However, each cache owns its content manager, replacement manager and resolution manager.

Query Cache: This cache uses Semantic Caching [3] to store the results of the previously executed queries.

Mobile Estimation Cache: This cache contains all the estimations corresponding to the query eventually processed on the mobile device. More specifically, it contains the estimated execution time and the energy consumption for the query result retrieval within the query cache.

Cloud Estimation Cache: This cache contains all the estimations corresponding to the query if it is processed on the cloud. Even though an estimation is found in this cache, it does not mean that the corresponding query has been processed. For both estimation caches, only the events of cache hit and cache miss can occur.

3.3 The MOCCAD Cache Algorithm

The MOCCAD-Cache algorithm shown in Fig. 4 consists of 4 main steps corresponding to the different phases occurring during a query processing algorithm: query cache analysis, estimation computation, decision making, and query evaluation. First we need to analyze the query cache in order to know what is usable to answer the input query. Then, depending on the result of the previous step, we may need to estimate the cost to process a query on the mobile device (processing time and energy consumption) as well as the cost (processing time, energy consumption and monetary cost) to process a query on the cloud. If we have to compute a mobile query processing estimation and a cloud query processing estimation, then we will need to make a decision upon those estimations and the user constraints in order to finally execute the query.

```
MOCCAD Cache Algorithm: CacheManager::load
Input: Query inQuery, QueryCache, MobileEstimationCache mobileEstiCache, CloudEstimationCache cloudEstiCache, MobileEstimationComputationManager mobileECompM, CloudEstimationComputationManager cloudECompM, OptimizationManager oM.
Output: Query Result.
1:   mobileEstiResult ← ∞
2:   cloudEstiResult ← ∞
4:   queryLookup,pQuery,rQuery ← queryCache.lookup(inQuery)
5:   if queryLookup = CACHE_HIT then
6:      mobileEstiResult ← 0
8:   else if queryLookup = PARTIAL_HIT then
9:      estiRemainder ← acquireEstimation(rQuery, cloudEstiCache, cloudECompM)
10:     estiProbe ← acquireEstimation(pQuery, mobileEstiCache, mobileECompM)
11:     mobileEstiResult ← estiRemainder + estiProbe
12:     cloudEstiCache ← acquireEstimation(inQuery, cloudEstiCache, cloudECompM)
14:  else if queryLookup = EXTENDED_HIT then
15:     mobileEstiResult ← acquireEstimation(inQuery, mobileEstiCache, mobileECompM)
16:     cloudEstiResult ← acquireEstimation(inQuery, cloudEstiCache, cloudECompM)
18:  else if queryLookup = CACHE_MISS then
19:     cloudEstiResult ← acquireEstimation( inQuery, cloudEstiCache, cloudECompM)
20:  end if
22:  bestEstimation, queryPlan ← oM.optimize(queryLookup, mobileEstiResult, cloudEstiResult)
23:  if bestEstimation != NIL then
24:     queryResult ← queryCache.process(queryPlan)
25:     queryCache.replace(queryResult, queryPlan)
26:  end if

acquireEstimation Function:
Input: Query query, EstimationCache estimationCache, EstimationComputationManager estimationCompManager
Output: Estimation estiResult.
1:   estiLookup ← estiCache.lookup(query)
2:   if estiLookup = CACHE_HIT then
3:      estiResult ← estiCache.process(query)
4:   else
5:      estiResult ← estimationCompManager.compute(query)
6:      estiCache.replace(query, estiResult)
7:   end if
8:   return estiResult
```

Fig. 4. MOCCAD-Cache Algorithm and acquireEstimation Function

First of all, in order to estimate a query processing cost, it is necessary to determine how much time and energy will be spent to analyze the query cache. The algorithm in [5] can be used to analyze two queries in order to know if they are equivalent or if one implies the other. This algorithm can be very expensive regarding the number of predicates contained in both the input query and the query in the cache. In order to estimate the time and energy necessary to process this algorithm, it is essential to compute some statistics for the execution time and the energy consumption regarding the number of predicates to be analyzed. This should be done only once when the application is started for the first time on the mobile device. We assume that the mobile device has a sufficient amount of energy to compute those statistics. Also, during the second phase of the estimation, we need some additional pre-processed data necessary to estimate the query processing cost on the mobile device. More specifically, we still need to download some metadata and information items from the database hosted on the cloud, which are required to estimate the size of the query result after we processed the query on it, such as relation name, number of tuples, average tuple size, etc. Those information items are downloaded once when the algorithm accesses the database. We assume that the database on the cloud is never altered.

This algorithm requires the query to be processed as an input. It needs three different caches: the query cache, the mobile estimation cache and the cloud estimation cache. It also uses the different managers for estimation computations and decisions.

First of all, we analyze the query cache (Line 4 in MOCCAD-Cache Algorithm). The query cache calls its content manager which will return 3 different items: the type of cache hit or miss, the probe query and the remainder query. Secondly, the estimation computation is made regarding the query analysis result (Line 5 to Line 20 in MOCCAD-Cache Algorithm). In the case of a query cache exact hit (*CACHE_HIT*), the query processing cost on the mobile device is considered negligible. Therefore, the query will be directly executed on the mobile device to retrieve the result. In the case of a partial hit (*PARTIAL_HIT*), it is necessary to estimate the processing time to process the probe query on the mobile device as well as the cost to process the remainder query on the cloud. Those two estimations need to be added to get the estimated cost to retrieve the complete result. Additionally, we need to compute the estimated cost to process the whole input query on the cloud. To acquire such estimations we use the acquireEstimation function. This function looks for those estimations in the cloud or mobile estimation caches (Line 1 in acquireEstimation Function). If they are not available in those caches, then the estimation is computed by the estimation computation manager. Then, the estimation cache calls its replacement manager to insert the new estimation into the cache (Line 4 to Line 6 in the acquireEstimation function). This way, even though the query is not executed, the estimation still belongs to the cache. This works the same way for both the cloud estimation cache and the mobile estimation cache. In the case of a cache extended hit (*EXTENDED_HIT*), it can be very expensive to process the query on the mobile device; it is therefore important to compute the estimation before this execution. Additionally, we estimate the costs regarding the execution of the query on the cloud in order to decide whether the query should be executed on the mobile device or on the cloud. Finally, in the case of a cache miss (*CACHE_MISS*), the algorithm computes the costs to process the query on the cloud to be sure they respect the user constraints.

Once the estimations have been computed, it is now possible to make a decision on where the query should be processed and then build the query plan (Line 22 in MOCCAD-Cache Algorithm). If it is possible to process the query while respecting the execution time, energy consumption and monetary cost constraints, then the query cache (*QueryCache*) calls its resolution manager to process the generated query plan and returns the result (Lines 23 to 26 in MOCCAD-Cache Algorithm). The execution time, the estimated energy and the money spent will be updated in the estimation caches to contain the real values. Then, we use the query cache’s replacement manager to replace the data within the query cache. If some segment should be removed from the query cache, the corresponding estimation will be removed from the mobile estimation cache. We cannot keep this estimation since it is based on the current segment on which the result can be retrieved. If this segment is replaced by another one requiring less processing cost than the first one to retrieve the result, then the estimation is not accurate anymore. However, another possible solution could be to store this estimation elsewhere for replacement purposes. Finally, after the replacement, the query result is sent to the user.

4 Experiments and Results

The MOCCAD-Cache algorithm and all the associated algorithms [5] and [4] have been implemented on Android Kit Kat 4.4.3. The experiments have been run on a HTC One M7ul embedding a Qualcomm Snapdragon 600 with a 1.728 GHz CPU, 2GB of RAM and a battery capacity of 2300 mAh. On the cloud side, a private cloud from the University of Oklahoma has been used. It uses one node with the following configuration: 16GB of RAM, Intel Xeon CPU E5-2670 at 2.60 GHz. A Hadoop framework (Version 2.4.0) as well as the data warehouse infrastructure, Hive, have been used for this experimentation. This cloud infrastructure can be accessed through a RESTful web service running on a tomcat server (Version 7.0). The web service can ask the cloud to estimate the cost of a query and to process a query on the cloud infrastructure using HiveQL. It can also return the metadata items related to the stored relation(s) using Hive Metastore. This Hive Metastore uses a MySQL Server (Version 5.1.73) to store the metadata. A 200,000 tuples relation is stored on HDFS and gathers 6 uniformly distributed attributes: ExpTable(bigint Id [200,000 values], string Name, string Info, int IntAttr1 [10,000 values], int IntAttr2 [50,000 values], int IntAttr3 [100,000 values]).

In order to study the performance of the MOCCAD-Cache in each case where we can use some results from the cache, the different experiments aim to measure the total query processing time, the total monetary cost and the total energy consumption for each percentage of exact hits, extended hits and partial hits on the query cache. This experiment is made on a query processor without cache, a query processor with a semantic cache, and a decisional semantic cache (MOCCAD-Cache). For each experiment, three runs have been done in order to minimize the environment noises such as Wi-Fi throughput variations, cloud computation time variation, and mobile processing time variation. Each run gathers 50 queries. The cache is warmed up with 10 segments storing from one tuple to 100,000 tuples. No cache replacement is used.

The experiment results show that the query processing time, the energy consumption and the monetary cost regarding the percentage of exact hits are similar between semantic caching and MOCCAD-Cache. This shows that our estimation cache prevents any overhead that could be caused by the estimation computation. Regarding the percentage of extended hits, when no cache is used, the query processing time (Fig. 5) decreases regarding the total result size of the used query sets. When MOCCAD-Cache is used, more queries are processed on the cloud compared to semantic caching. Therefore, the different estimations and decisions occurring when using the MOCCAD-Cache determine that it is more efficient to process a query on the cloud rather than on the mobile device. Thus, our MOCCAD-Cache algorithm is faster than semantic caching. However, more money is spent on the cloud since more queries are processed on it (Fig. 6). When a money constraint is added, our algorithm can prevent too expensive queries from being processed. Therefore, the number of queries not meeting the constraints is reduced with our algorithm compared to semantic caching which does not take into account any of the user constraints.

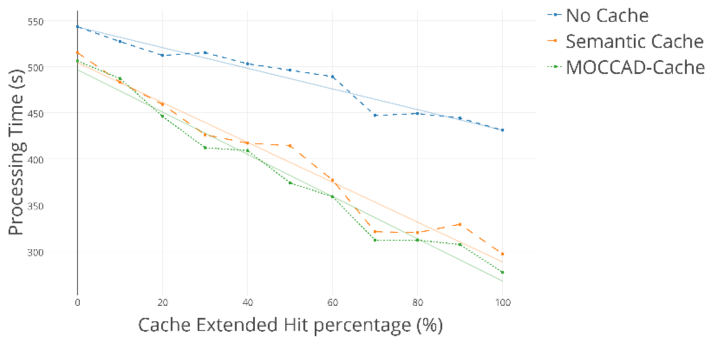


Fig. 5. Processing Time for Fifty Queries Regarding Cache Extended Hit Percentage

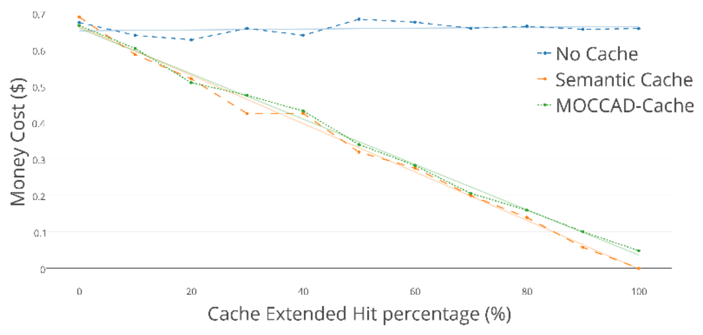
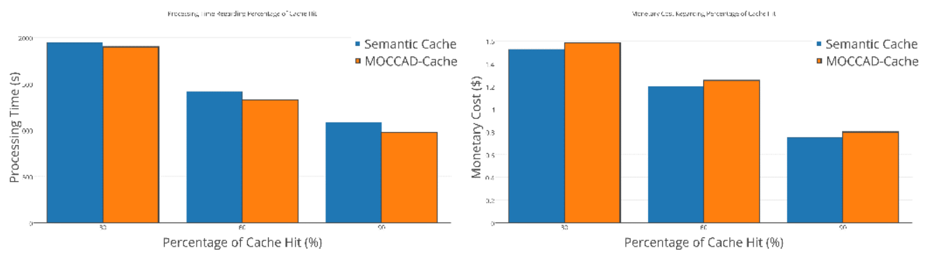


Fig. 6. Monetary Cost for Fifty Queries Regarding Cache Extended Hit Percentage

Regarding the partial hit percentage, our algorithm performs similarly to semantic caching. When the cloud can perform queries quickly with a high number of cloud instances, the estimated time and energy are mainly related to the download time and the size of the query results. Thus when a decision needs to be made, the algorithm chooses to process the query retrieving the smallest amount of data. This corresponds to the query plan which processes the probe query on the mobile device and the remainder query on the cloud. This query plan is identical to the one used in semantic caching and thus, no improvement is being made.

Finally, our algorithm is validated by showing the impact of MOCCAD-Cache on the query processing time and the monetary cost (Fig. 7). We compare our solution with semantic caching. Those results correspond to the total of the previously presented results. Thus each value of time and monetary cost corresponds to 150 processed queries with the same percentage of exact hit queries, extended hit queries and partial hit queries. The most significant and relevant values are presented. Fig. 7.a shows that the query processing time is globally reduced when we use MOCCAD-Cache. Fig. 7.b shows that the user will however need to pay the price of this improvement in terms of time. Once again, the small difference in time and money cost between semantic caching and our algorithm is due to the small cloud that has been used (only 5 instances). Additional work would be required to process the same experimentation on a bigger cloud (with more instances) to really emphasize this difference.



(a) (b)

Fig. 7. Global Impact of Cache Hit Percentage on the Query Cost

5 Conclusion and Future Work

In this research, we have proposed a time, energy and money cost-aware semantic caching algorithm called MOCCAD-Cache for a 3-tier mobile cloud database architecture. MOCCAD-Cache can estimate the time, energy and money to be spent for a given query when it is processed on the cloud or when it is processed on the mobile device. This way, it can decide which query plan is the best considering the user constraints in terms of query processing time, energy consumption and monetary costs, and is an improvement over semantic caching because it can perform queries faster in a Mobile-Cloud database environment. However, the user would have to pay more monetary cost consequently.

MOCCAD-Cache is the first decisional semantic caching algorithm for a Mobile-Cloud database system. This first step of optimization and decision making within a caching algorithm on the mobile device needs many expansions and opens many possibilities. A future step aims at handling several cloud providers and chooses to process a query only on the services where it is the least expensive regarding the user constraints.

References

1. P. Mell et T. Grance, «The NIST definition of cloud computing,» *NIST,* vol. 53.6, p. 50, 2009.
2. A. Delis et N. Roussopoulos, «Performance and scalability of client-server database architectures,» *VLDB*, 1992.
3. S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava et M. Tan, «Semantic data caching and replacement,» *VLDBJ,* vol. 96, pp. 330-341, 1996.
4. Q. Ren, M. H. Dunham et V. Kumar, «Semantic caching and query processing,» *IEEE Transactions on nowledge and Data Engineering,* vol. 15.1, pp. 192-210, 2003.
5. S. Guo, W. Sun et M. A. Weiss, «Solving satisfiability and implication problems in database systems,» *ACM Transactions on Database Systems (TODS),* vol. 21, n° %12, pp. 270--293, 1996.
6. M. A. Abbas, M. A. Qadir, M. a. A. T. Ahmad and N. A. Sajid, "Graph based query trimming of conjunctive queries in semantic caching," *IEEE 7th International Conference on Emerging Technologies (ICET), 2011* , 2011.
7. M. Ahmad, S. Asghar, M. A. Qadir et T. Ali, «Graph based query trimming algorithm for relational data semantic cache,» *the ACM International Conference on Management of Emergent Digital EcoSystems,* 2010.
8. M. Ahmad, M. Qadir et M. a. B. M. Sanaullah, «An efficient query matching algorithm for relational data semantic cache,» *IEEE 2nd International Conference on Computer, Control and Communication, 2009*.
9. B. Chidlovskii et U. M. Borghoff, «Semantic caching of Web queries,» *VLDBJ,* vol. 9.1, pp. 2-17, 2000.
10. B. Þ. Jónsson, M. Arinbjarnar, B. Þórsson, M. J. Franklin et D. Srivastava, «Performance and overhead of semantic cache management,» *ACM Transactions on Internet Technology (TOIT),* vol. 6.3, pp. 302-331, 2006.
11. Q. Ren et M. H. Dunham, «Using semantic caching to manage location dependent data in mobile computing,» *the 6th annual international conference on Mobile computing and networking*, 2000.
12. K. C. Lee, H. V. Leong et A. Si, «Semantic Query Caching in a Mobile Environment,» *ACM SIGMOBILE Mobile Computing and Communications Review,* vol. 3.2, pp. 28-36, 1999.
13. A. Silberschatz, H. F. Korth et S. Sudarshan, Database system concepts, McGraw-Hill New York, 1997.
14. A. Carroll et G. Heiser, «An Analysis of Power Consumption in a Smartphone,» *USENIX annual technical conference*, 2010.
15. M. Gordon, L. Zhang, B. Tiwana, R. Dick, Z. Mao et L. Yang, *Power Tutor, A Power Monitor for Android-Based Mobile Platforms,* 2009.
16. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein et others, Introduction to algorithms, MIT press Cambridge, 2011.
17. L. d'Orazio, «Caches adaptables et applications aux systèmes de gestion de données reparties a grande échelle,» 2007.