UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

TIME-, ENERGY-, AND MONETARY COST-AWARE CACHE DESIGN FOR A MOBILE-CLOUD DATABASE SYSTEM

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

MIKAEL PERRIN
Norman, Oklahoma
2015

TIME-, ENERGY-, AND MONETARY COST-AWARE CACHE DESIGN FOR A MOBILE-CLOUD DATABASE SYSTEM

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Le Gruenwald, Chair

Dr. Laurent d’Orazio

Dr. Qi Cheng

Acknowledgements

I particularly thank Dr. Le Gruenwald, the chair of my thesis's committee and my supervisor at the University of Oklahoma, as well as Dr. Laurent d'Orazio, my supervisor at the University of Blaise Pascal and committee member, for their availability during this project, their guidance all along the different thesis steps and for making me understand the ins and outs of the research area. I would like to thank Dr. Qi Cheng for his interest in my thesis project and for taking part on my research committee.

I thank Mr. Jonathan Mullen for his help on many technical challenges and for his contribution during this project. I also thank Mr. Romain Perriot and Mrs. Baraa Mohamad for their availability and their help during this project.

I thank Mrs. Virginie Perez Woods for her help within the partnership between the ISIMA (Institut Supérieur d'Informatique, de Modélisation et de leurs Applications) and the University of Oklahoma.

I thank the National Science Foundation (NSF) for partially funding my thesis.

Also, I thank Mr. Thibault Lucidarme, Mr. Cyril Beyney, Mrs. Olivia Perret and Mr. Jeremy Pfeifer for their friendship and their support during those two years at the University of Oklahoma. Finally, I thank my sister and brother-in-law, Magali and Cyril Bourgin, and my parents, Annick and Raymond Perrin for their support, without whom this unbelievable experience would not have happened.

This material is based upon work supported in part by the National Science Foundation under Grant No.1349285. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

List of Tables

List of Figures

Abstract

Growing demand for more mobile access to data is only matched by the growth of large and complex data. The availability and scalability of cloud resources when combined with techniques of caching and distributed computation provide tools to address these problems, but bring up new multi-dimensional optimization challenges concerning the three cost constraints: query execution time, energy consumption on the mobile device, and monetary cost to be paid to the cloud service providers. To address these challenges, this thesis develops a caching algorithm, called MOCCAD-Cache, for a MObile Cloud Cost-Aware Database system (MOCCAD) that consists of three tiers: mobile users, data owner, and cloud. The algorithm is aimed at improving query response time while taking the constraints on energy consumption and monetary cost into consideration.

To improve query response time, MOCCAD-Cache constructs a *query cache* on the mobile device based on the concept of semantic caching. This cache stores not only the results of the previous queries, but also the metadata associated with those queries. With this metadata, further server communication is either eliminated when the result of a query is stored entirely in the cache, or reduced when the cache contains only a part of the data required by the query. To determine this, MOCCAD-Cache employs a query trimming technique to compare the input query with the query contained in each metadata entry. In the case when the cache partially contains the query result, a probe query will be created to retrieve the existing data from the query cache, and a remainder query will be created to retrieve the data not contained in the query cache. The remainder query will then be sent to the cloud for execution.

Introducing semantic caching makes it mandatory to find a trade-off between running the query on the cloud and running the query on the mobile device with respect to the three constraints. In particular, it is necessary to estimate the time and energy to be spent to retrieve data on the mobile device and to compare them with the time and energy to retrieve data on the cloud. Since computing these estimations requires time, energy and money, MOCCAD-Cache equips the mobile device with an *estimation cache* to reduce such overheads. Each estimation cache entry contains the estimated time and energy required to process a query on the mobile device as well as the estimated time and monetary cost needed to process the same query on the cloud. MOCCAD-Cache then uses those estimations to determine which one, the mobile device or the cloud, is the best one to execute a query given the query's constraints. Finally, it constructs an appropriate plan to execute the query.

To study the performance of the proposed algorithm, a prototype was built on an Android Smartphone connected to a private cloud instance where the database system, Hive, was used to manage data. The prototype allows the user to build queries, specify constraints, retrieve the query results, examine the query cache and the estimation cache, and view statistics about the query executions in order to analyze the query processing performance. Experiments were conducted on the prototype using various queries.

Chapter 1: Introduction

1.1 Definitions

1.1.1 Mobile Environment

While not a new concept, mobility is a field found in more and more research problems, both in academia and in industry. The main principle of mobile computing is the possibility to provide to the user resources independent of his/her location. The user does not need to be at home, in front of a computer, to access data. A mobile device has the advantage of being light and easily transportable, but nevertheless, has limited resources. Indeed, memory, computing power, screen size, network connection and energy are several constraint examples that need to be considered within the mobile computing field.

1.1.2 Cloud Environment

The constraints associated with a mobile environment discussed in the previous section 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. Even though the client-server architecture is well understood, it does not provide the advantages of cloud computing, such as elasticity and high end resource management. Indeed, when a query is sent to a service on the cloud, this service can either decide to add a number of nodes to process the query (scale out or horizontal elasticity) [1], or to increase the computation power and/or the memory of each node processing the query (scale up or vertical elasticity) [1]. Also, even though cloud computing growth shows different obstacles such as data business continuity, data lock-in or bugs in large distributed database systems, it also provides different opportunities such as using multiple cloud providers, providing standardized APIs or inventing a debugger relying on distributed VMs for example [2].

1.1.3 Mobile-Cloud Environment

The previous subsections can show how interesting it can be to provide cloud computing services to mobility. Undeniably, when the mobile device's lack of resources becomes restrictive, using cloud services becomes necessary to answer to the user constraints. This leads to what has been defined as Mobile Cloud Computing which has many possible applications such as image processing, natural language processing, sharing GPS, sharing internet access, sensor data applications, crowd computing, and multimedia search and querying [3]. In this type of environment, the mobile device can be considered as a client (i.e. the Gmail application) using some powerful servers (i.e. Google Servers), and providing some additional information such as client's location. Another possibility is to use the mobile devices as resources. An example to this is a mobile device running a language translation application on a local cloud. Other mobile devices act as resource providers. Finally, [4] defines another architecture offloading the work on several local multi-core computers, called cloudlet, and being connected to the cloud servers. Those multi-core computers allow the user to have a high performance connectivity with the cloud and thus, permit some applications such as real-time image recognition.

1.1.4 Cache

To reduce the amount of communication throughout the network between the mobile device and the server (or a cloud service), a cache system can be used [5]. A cache system is a software or a hardware component allowing to store previously processed data. Those data could have been computed or downloaded from a server. If those data were to be requested again, the cache would return them in a transparent way in order to minimize the processing cost of the queries. At least two different event types can occur in the cache:

- Cache hit: A looked up item is found within the cache.
- Cache miss: A looked up item cannot be found within the cache.

Several caching types exist according to the application domain for which they have been used (CPU caching, Web caching, DBMS caching), or according to the used algorithm, for example, a database management system can contain tuple caching, page caching or semantic caching among other things [6]. The cached data can be stored within different types of memories (virtual memory or physical memory) and can be handled directly by the hardware (CPU caches).

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. The first one will be used to retrieve data from the cache. The other one will be sent to the server to retrieve the missing data. To illustrate this type of caching, let us consider a scenario in which a doctor wants to retrieve the data of his/her patients being 30 years old or younger. Within the cache, all the data items of the patients 32 years old or younger can be found. Thus, the cache already contains the doctor’s query result. With a semantic cache, the segments will be analyzed and then processed to retrieve this result from the cache. However, with a standard cache, the doctor’s query and the query contained in the cache are different. Therefore, the doctor’s query will be processed on the database server.

1.2 Problem Definition

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 results. Therefore, this adds up a third dimension to the optimization problem: the mobile device's energy consumption. As explained earlier in Section 1.1.4, when using a cache system, the main goal is to reduce the network communication between the user and the cloud, and therefore, the query processing time. In addition, using semantic caching allows the size of the data transferred between the cloud and the mobile device to be reduced. However, in order to determine whether it is possible to use one or several entries in the cache, it is necessary to analyze those cache entries beforehand. This analysis has a cost that can impact the energy consumption more than the query execution time [7].

1.2.1 Existing Architecture

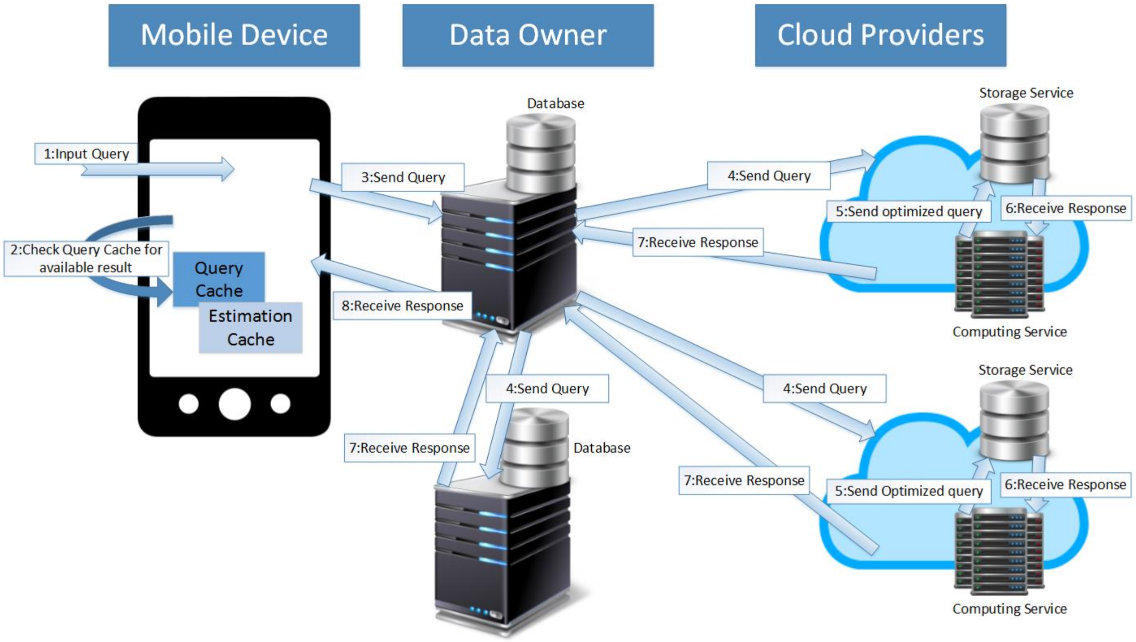


Figure 1 - 3-tier architecture

To be able to solve this optimization problem in an environment where data locality matters, a 3-tier architecture has been modelled (Figure 1). This architecture was originally proposed for 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 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. Those resources are not elastic: adding a new server will require additional money and time mandatory for its purchase, delivery, set up and configuration. This static infrastructure is called the data owner within our architecture. To finish, some of the information is not considered private, like this meeting’s date and time with the director, for example. These information items can be stored on elastic infrastructures on the cloud called service 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. While this architecture was originally developed for medical applications, it is general enough to be applied to many other applications. Below we describe the three tiers of the architecture: the mobile device, data owner, and cloud service provider.

The mobile device: it is used to build queries that allow mobile users to access data stored either at the data owner or at the cloud, thanks to the application user interface. The device is easily transportable, but does not own many resources. The application uses a query cache to store the user's query results and the query’s metadata.

The data owner: it owns the data items and is responsible for them. In France, a certified medical host is used for it. In the United States, the medical information does not have to be stored in a certified medical host; however, several rules defined by the HIPAA (Health Insurance Portability and Accountability Act) [8] makes a minimum storage security mandatory for the different cloud services.

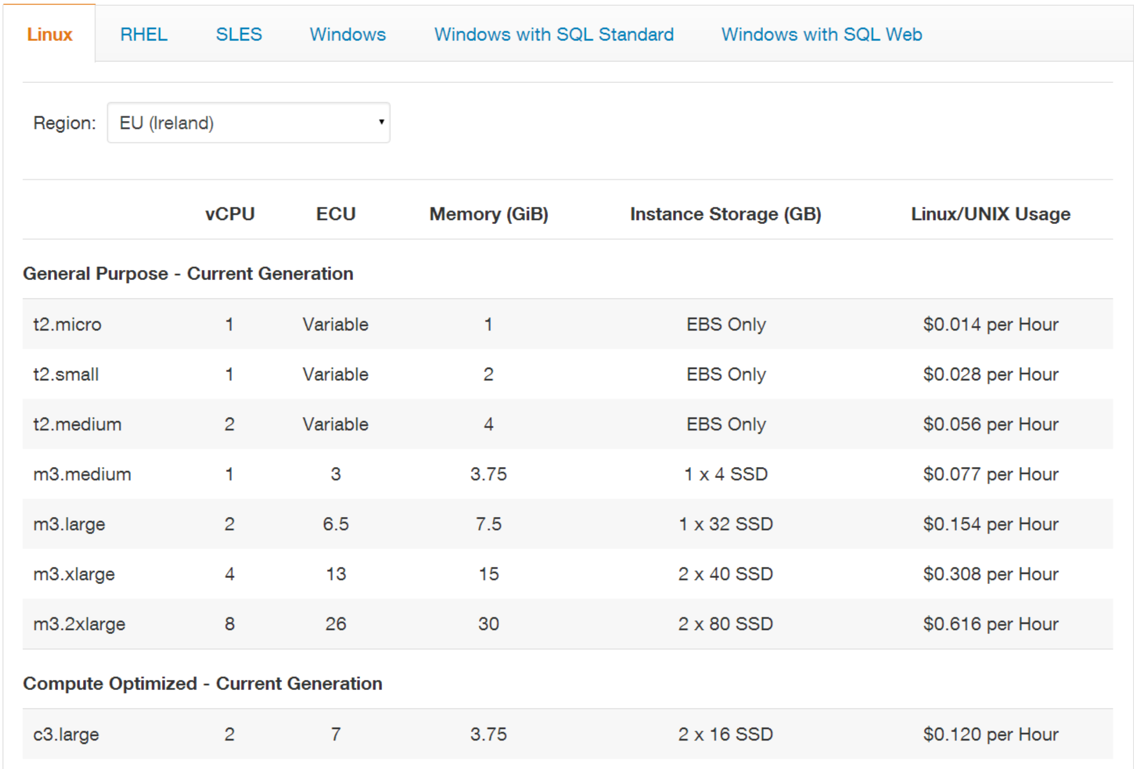
The Cloud service provider: It provides highly scalable services to its users’ needs. [1] and [2] define the different terms used to refer to cloud computing. Cloud computing refers to the applications as services and the infrastructure (software and hardware) providing those services. There exist different levels of services on the cloud: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [1]. Each service provider provides several pricing models regarding the types of services, the used resources and the server locations. For example, the Amazon's computation service (EC2) gives the details of its prices for each usable services on its environment (Figure 2). A public cloud is defined as a data center (hardware and software) providing a service called utility computing. This service is based on the pay-as-you-go method which charges the user based on usage. A private cloud is defined as an internal data center of an organization made available privately and large enough to be used for cloud computing. Concerning pricing, cloud computing infers that the resources are available on demand and quickly enough to adapt to any escalation. Also, cloud users do not have any commitment, meaning that users can use more resources only when there is an increase in their needs and can stop paying as soon as they do not want to use those services anymore. To define the cost related to each application, cost models in terms of computation, storage and communication are necessary. [9] provides examples of cost models and different challenges related to materialized views in the cloud. Indeed, such views require some cost for storage, computation and communication and thus emphasize the tradeoff between time and monetary cost we can find on the cloud. For example, Table 1 and Table 2 show some examples of costs when one respectively uses and does not use a materialized view on the cloud. With the costs in Table 1, the storage cost is $50 and the computing cost is $12. With the costs in Table 2, the storage cost is $55 and the computing cost is $9.6. With materialized views, the cost performance increases by 20% but the price increases by 4%. These examples help to see the types of tradeoffs between time and money that we can find when dealing with cloud computing.

Table 1 - Example of the cost without materialized view

Storage Cost	$0.10/GB/Month
Computing Cost	$0.24/hour
Dataset size	500GB
Monthly Processed Queries	50 hours

Table 2 - Example of the cost with a materialized view

Storage Cost	$0.10/GB/Month
Computing Cost	$0.24/hour
Dataset size	550GB
Monthly Processed Queries	40 hours



Region: EU (Ireland)

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
t2.micro	1	Variable	1	EBS Only	$0.014 per Hour
t2.small	1	Variable	2	EBS Only	$0.028 per Hour
t2.medium	2	Variable	4	EBS Only	$0.056 per Hour
m3.medium	1	3	3.75	1 x 4 SSD	$0.077 per Hour
m3.large	2	6.5	7.5	1 x 32 SSD	$0.154 per Hour
m3.xlarge	4	13	15	2 x 40 SSD	$0.308 per Hour
m3.2xlarge	8	26	30	2 x 80 SSD	$0.616 per Hour
Compute Optimized - Current Generation					
c3.large	2	7	3.75	2 x 16 SSD	$0.120 per Hour

Figure 2 - Amazon Web Service prices - EC2

In the context of the hospital example, once the director has chosen the doctor for whom he wants to retrieve the information, the mobile device application will first determine whether the cache stores a part of the result of the query. If some of the information items that need to be retrieved in addition to those available are in the cache, then the query is sent to the data owner. Those information items regarding the doctor can be located on other data owners' servers that are willing to share those items with the current data owner or on some storage services on the cloud. Therefore, it is necessary to send the query to those other data owners and cloud services in order to retrieve the whole query result (i.e. all the information items concerning the doctor and his/her projects).

1.2.2 Problem

Within the previously described environment, we can therefore understand this thesis problem which involves the following research questions: *How can we process fast queries from a mobile device while respecting the constraints on energy consumption and monetary cost? How to perform cheap queries while respecting the constraints in energy and time? 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 thesis is the development of a second cache on the mobile device called *estimation cache* working with the semantic query cache. To solve this complex problem, it is mandatory to start with a simpler version of the problem. First, several assumptions are made about the type of query posed by the user. Indeed, we suppose that each query is processed against one relation only, the goal being not having to handle join operations. Also, this query will be a selection query without any projection. This way, if a tuple is stored in the cache, we know it owns all the attributes of the relation. The query predicates will correspond to a conjunction of inequalities. Concerning the network connection between the mobile device and the data owner, we assume that it will always be sufficient, that the wireless technology used is Wi-Fi and that no network interruption happens during the data transfer between the two entities. The time and money used by the data owner to estimate the cost for the query evaluation on the cloud servers or on the data owner itself will be considered negligible. Also, we will assume that the database hosted on the cloud or on the data owner 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 the cache, the data owner and the cloud servers will be considered as one. For a query cache hit, the time, energy and money used are also considered negligible, and the query will also be automatically evaluated and its result will be sent back to the user.

1.3 Organization of the Thesis

First of all, this thesis details the background and literature review necessary for the understanding and the elaboration of a first solution in Chapter 2. This part describes the different caching possibilities and the different estimation techniques used during query processing. Secondly, this thesis presents the proposed algorithm, called MOCCAD-Cache, to solve the introduced problem as well as the solution’s architecture in Chapter 3. Then, Chapter 4 describes the solution’s prototype implementation and the results of the experiments conducted to study the performance of the proposed solution. Finally, Chapter 5 concludes the thesis and presents future work.

Chapter 2: Literature Review

This chapter reviews the existing work in the areas of semantic caching and query processing estimations. In addition, it details the different tools needed to build the proposed MOCCAD-Cache algorithm.

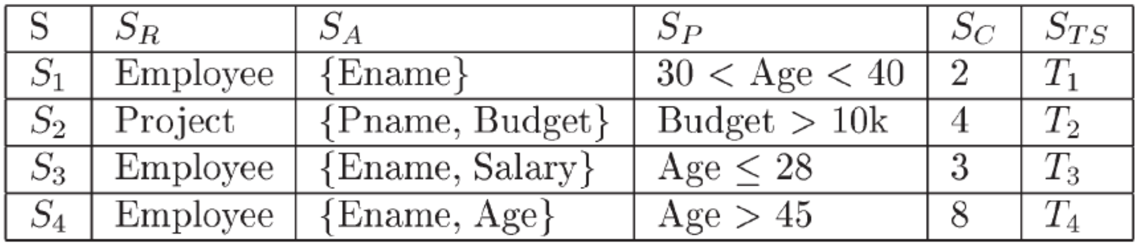
2.1 Semantic caching

In order to demonstrate the performance of semantic caching for a server in ideal conditions, [6] and [10] compared this type of caching with page caching and tuple caching. A page is a transfer unit of a static size between the client and the server. For page caching, when a query is posed at the client side, if one of the pages has not been found, then all the data corresponding to this page will be obtained from the server. For tuple caching, the unit is the tuple itself. This type of cache brings up good flexibility but a huge overhead in terms of performance. More recently, [11] showed the performance of semantic caching for some types of applications requiring important workloads and using a limited bandwidth environment. This type of environment specifically can be very similar to the one in our application domain since medical data gathers many information items which make their processing difficult and expensive and since the bandwidth on the mobile device is unstable due to its owner’s displacement. Therefore, those papers support the choice of semantic caching inside our Mobile-Cloud environment as a good way to reduce the overhead cost of query processing.

[12] and [13] studied semantic caching performance in a mobile environment and mainly focus on location dependent applications to emphasize how useful semantic caching can be in such an environment. Indeed, location dependent semantic caching

brings another step to this technique since the data in the cache can be sorted and replaced, not only based on its access time but also based on its location. [10] proposes algorithms allowing semantic caching to be used for web queries to retrieve relevant information efficiently in Web repositories.

[6] and [14] offer a formal definition of semantic caching. Three key ideas are used in defining semantic caching. First of all, 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, it contains a value function permitting the replacement of the cache data. For example, if the chosen value function is time, we can choose to replace the oldest entries. Finally, each entry contains a set of tuples storing the result of each query. The size of this set is dynamic. Each entry is called a semantic region and is considered as the unit for replacement within the cache. It stores a list of projection attributes, a list of 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. Figure 3 shows an example of a semantic cache with S the segment id, S_R the relation from which the tuples come from, S_A the projection attributes, S_P the predicates, S_{TS} the time stamp used for replacement, and S_C the content of the segment or more exactly the location of the tuple set.



S	S_R	S_A	S_P	S_C	S_{TS}
S_1	Employee	{Ename}	$30 < \text{Age} < 40$	2	T_1
S_2	Project	{Pname, Budget}	$\text{Budget} > 10\text{k}$	4	T_2
S_3	Employee	{Ename, Salary}	$\text{Age} \leq 28$	3	T_3
S_4	Employee	{Ename, Age}	$\text{Age} > 45$	8	T_4

Figure 3 - Semantic Cache Example [14]

When a query is processed, the semantic cache manager analyses the cache content and creates two sub-queries. The first one, called *probe query,* is used to retrieve data in the cache. The second one, called *remainder query*, retrieves the missing data from the database server. [14] defines the notion of *query trimming* which is the process corresponding 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 [15], [16] and [17]. However, even though their contributions improve the performance of the query trimming algorithm, the version in [14] provides much more accuracy towards its implementation. Indeed, with the contribution of this thesis lying mainly in the estimation component, it is therefore important to choose a reliable and easy-to-implement semantic caching system for the query cache. During this query trimming algorithm, 4 types of event can occur: cache hit, cache extended hit, cache partial hit, and cache miss. These events are described in detail below.

Cache Hit: As previously described, it defines the availability of a complete query result in the cache. The following illustration (Figure 4) presents an example of this type of event for the selection of patients with a heart rate lower than 57 from the database table *NOTE* where the heart rate is represented by the attribute *HR*. Query (1) is the same as the input query; therefore we can use its result to respond to the input query.

Input query: $\sigma_{HR<57}(NOTE)$	*Queries available in the query cache:* $\sigma_{HR<57}(NOTE)$ *(1)* $\sigma_{age>28 \wedge age<32}(DOCTOR)$ *(2)* $\sigma_{HR=57}(NOTE)$ *(3)*

Figure 4 - Example: Cache Hit

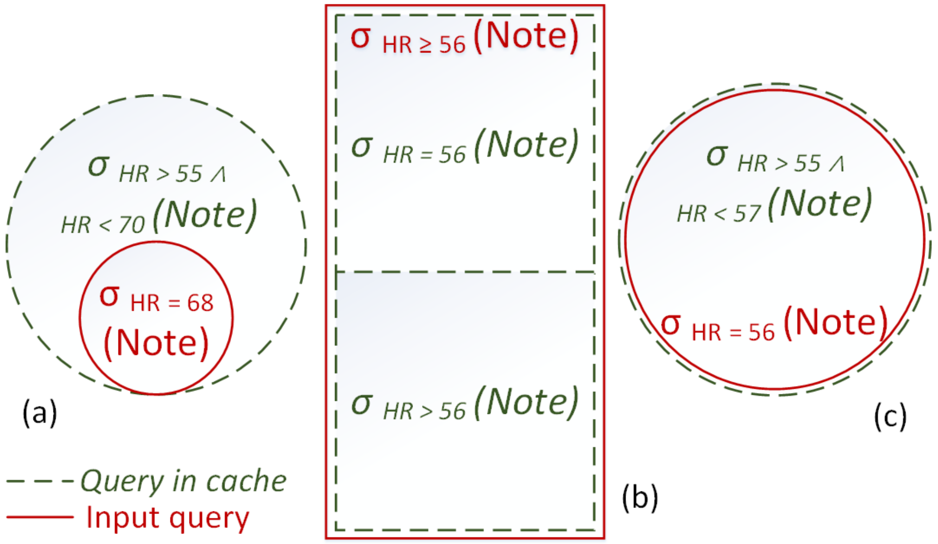


Figure 5 – Example: Cache Extended Hit

Cache Extended Hit: Some additional operations and processing need to be done in order to retrieve the whole query result from the cache. There are several types of cache extended hit as shown in Figure 5. Figure 5.a shows a case where the result of the input query is included inside one of the regions in the cache. Figure 5.b shows a situation where the result of the input query needs to be retrieved from several segments in the cache. Then, Figure 5.c shows a case where the input query and the semantic region’s query are equivalent. 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. Indeed, in the following example (Figure 6), query (3) can be used in order 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.

Input query: $\sigma_{HR\geq 57}(NOTE)$	*Queries available in the query cache:* $\sigma_{HR<57}(NOTE)$ *(1)* $\sigma_{age>28 \wedge age<32}(DOCTOR)$ *(2)* $\sigma_{HR=57}(NOTE)$ *(3)*

Figure 6 - Example: Cache Partial Hit

Cache Miss: This type of event represents the case where none of the semantic regions can be used to answer to the input query. Figure 7 emphasizes this statement by showing an example where none of the tuples corresponding to 42 year old doctors required for the input query is present in the cache.

Input query: $\sigma_{age=42}(DOCTOR)$	*Queries available in the query cache:* $\sigma_{HR<57}(NOTE)$ *(1)* $\sigma_{age>28 \wedge age<32}(DOCTOR)$ *(2)* $\sigma_{HR=57}(NOTE)$ *(3)*

Figure 7 - Example Cache Miss

2.1.1 Query Trimming

In order to know which type of query trimming event happens, it is not necessary to process all the tuples within each semantic region. Indeed, a semantic cache provides a sufficient amount of information to be able to compare the input query and the semantic region's query and thus, gain performance. [18] provides such an efficient algorithm to solve the problems of implication and satisfiability for a conjunction of inequalities as defined below. These concepts are used in [14] for query trimming. The main idea is to be able to compare the input query predicates with the current semantic region's query predicates.

An inequality, as defined in [18], can have two different shapes: {*X **op** C}* or *{X **op** Y}*, where *X* and *Y* are attributes or variables which belong to the real or integer domain, *C* is a constant which belongs to the same domain as *X*, and *op* is the comparative operator in {<, ≤, =, ≥, >, ≠}.

Satisfiability [18]**:** The predicates (in this situation, a conjunction of inequalities) are said to be satisfiable if there exists an assignment returning true for the values assigned to the different attributes. For example, the following set of predicates (1) is satisfiable because the assignment *X = 21* and *Y = 12* returns true.

$$S = (X > 12) \wedge (X < 22) \wedge (Y = 12) \qquad (1)$$

Also, the following set of predicates (2) is satisfiable if $X \in \mathbb{R}$, but is unsatisfiable if $X \in \mathbb{Z}$. Indeed the assignment *X = 10.5* returns true.

$$S = (X < 12) \wedge (X > 10) \wedge (X \neq 11) \qquad (2)$$

Implication [18]**:** A conjunction of inequalities S implies a conjunction of inequalities T if each assignment satisfying S also satisfies T. For example, S implies T in the following:

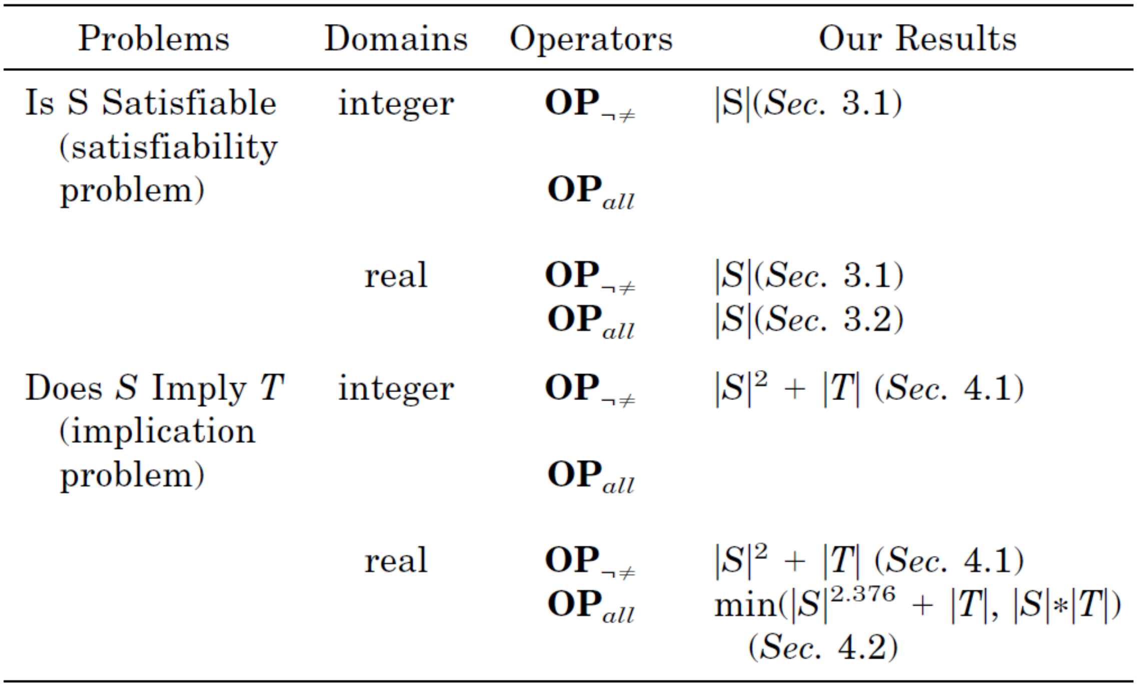
$$S = (X > 0) \wedge (X < 4), X \in \mathbb{Z} \tag{3}$$

$$T = (X \geq 0) \wedge (X < 5), X \in \mathbb{Z} \tag{4}$$

Indeed, to satisfy S in the integer domain, $X \in \{1,2,3\}$. Since the assignment of any of those values returns true for T, we can say that S implies T.

Equivalence [18]**:** If S implies T and T implies S, S and T are equivalent.

The complexity of each algorithm introduced by [18] is shown in Figure 8. |S| and |T| represent the numbers of inequalities for the set of predicates S and the set of predicates T, respectively. The operator $OP_{\neg\neq}$ indicates that no inequality can be found with the operator ≠. The operator OP_{all} indicates that each inequality operator belongs to the set of operators {<, ≤, =, ≥, >, ≠}.



Problems	Domains	Operators	Our Results
Is S Satisfiable (satisfiability problem)	integer	$\mathbf{OP}_{\neg\neq}$	$\lvert\mathrm{S}\rvert(Sec.\ 3.1)$
		$\mathbf{OP}_{all}$	
	real	$\mathbf{OP}_{\neg\neq}$	$\lvert S\rvert(Sec.\ 3.1)$
		$\mathbf{OP}_{all}$	$\lvert S\rvert(Sec.\ 3.2)$
Does S Imply T (implication problem)	integer	$\mathbf{OP}_{\neg\neq}$	$\lvert S\rvert^2 + \lvert T\rvert\ (Sec.\ 4.1)$
		$\mathbf{OP}_{all}$	
	real	$\mathbf{OP}_{\neg\neq}$	$\lvert S\rvert^2 + \lvert T\rvert\ (Sec.\ 4.1)$
		$\mathbf{OP}_{all}$	$\min(\lvert S\rvert^{2.376} + \lvert T\rvert, \lvert S\rvert{*}\lvert T\rvert)$ $(Sec.\ 4.2)$

Figure 8 - Satisfiability and Implication Algorithms Complexity [18]

However, this algorithm brings up an additional constraint towards the type of attribute that can be used within the cache. Indeed, this algorithm only uses integer or real numbers within a segment, but not both.

2.1.2 Query Processing

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. The following example shows the limitations of using the semantic cache algorithm.

Let's assume that one user wants to process the following input query:

$$Q_1 = \sigma_{id=42}(DOCTOR)$$

The cache already contains the result for the query:

$$Q_S = \sigma_{id<200000}(DOCTOR)$$

In order to retrieve the input query result on the mobile device, 199999 tuples need to be analyzed (in the case there are no indexes) when only one tuple corresponding to the doctor id number 42 needs to be retrieved. Processing Q_1 on the cloud which is able to process the query way faster, may be relevant in this case, especially because the time to retrieve the result through the network is derisory since it corresponds to the retrieval of only one tuple.

In addition, there might be a similar limitation in the case of a partial hit on the query cache. Let's take the following queries as an example:

Let Q_2 be the query that some user wants to process.

$$Q_2 = \sigma_{id \geq 150000 \wedge id < 150010}(DOCTOR)$$

And Q_S the query contained in the cache:

$$Q_S = \sigma_{id \leq 150000}(DOCTOR)$$

In this case only the tuple corresponding to the id equal to 150000 can be retrieved from the cache. Thus, when this type of partial hit occurs, the entire segment needs to be processed to retrieve the one and only tuple answering Q_2, and in addition, it needs to retrieve the remaining part of the result from the Cloud. The limitation is quite stark in this case since processing the whole query on the cloud can be way faster, especially to retrieve only ten tuples.

2.2 Optimization

2.2.1 Query estimation

One of the main steps during query processing is called query optimization. It comes after the query translation and before the query evaluation. Given a query, there are several ways to retrieve its result. For example, if a query involves a selection and projection, it could either be decided to apply the projection before the selection or to apply the selection before the projection. Choosing which query is less expensive to evaluate is part of this query optimization process. Therefore, there exist several evaluation plans for a query. To be able to choose one, cost estimations are required. For example, if the goal is only to minimize the money spent on the cloud, the choice should be on the query that has been estimated to consume the minimum amount of money on the cloud. [19] explains that the execution cost for an operation on a database depends on the statistics and the sizes of the inputs. In order to compute the total amount of time and energy needed to process the query on the mobile device 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 which is sent back from the cloud. [20] provides a formula to compute this size. This formula provides the following items:

- n_r, the number of tuples in the relation r (retrieved from the database catalog on the cloud).
- I_r, the size of each tuple in the relation r (retrieved from the database catalog on the cloud, with the assumption that the database will not be altered).
- *V(A,r)*, the number of different values that we can find in the relation *r* for the attribute *A*. This value is equivalent to the size of $\pi_A(r)$. If *A* is the primary key of the relation *r*, $V(A,r) = n_r$ (retrieved from the database catalog on the cloud).

If the query corresponds to $\sigma_{A=a}(r)$, the size can be computed as follows:

$$Size = I_r \times \frac{n_r}{V(A,r)} \tag{5}$$

If the query corresponds to $\sigma_{A\leq v}(r)$, the size can be computed as follows:

$$Size = 0, if\ v < \ \mathrm{min(A,r)} \tag{6}$$

$$Size = n_r \times I_r, if\ v > \ \mathrm{max(A,r)} \tag{7}$$

$$Size = I_r \times \frac{n_r \ \times (\ v - \min(A, r) \ )}{(\ \max(A, r) - \min(A, r) \ )}, otherwise \tag{8}$$

$\min(A, r)$ and $\max(A, r)$ are respectively the minimum value and the maximum value in the relation r for the attribute A.

2.2.2 Time estimation

Knowing the bandwidth speed between the cloud and the mobile device, we can determine the time to transfer data between the two entities. For example, if the query result size is estimated to be 2 MB, and the throughput between the mobile device and the cloud is around 32 Mb/s (4 MB/s), then the time to retrieve the result is estimated to be 0.5 seconds.

Finally the estimated time to process the query on the cloud $t_{process\_cloud}$, is:

$$t_{process\_cloud} = t_{send\_query} + t_{exec\_query} + t_{send\_result} \tag{9}$$

With

- t_{send_query}: the time to send the query to the cloud. This time is negligible due to the size of the data sent (only a few bytes).
- t_{exec_query}: the time to process the query on the cloud.
- t_{send_result}: the time to download the query result from the cloud to the mobile device.

2.2.3 Energy estimation

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:

- $t_{compute\_estimation}$, 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).
- $t_{process\_mobile}$, the time to process a query on the mobile device.
- $t_{process\_cloud}$, the time to process a query on the cloud.

Also, different steps described in [7] and [21] 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}, the energy consumed when the mobile device is awake but no application is active (considered negligible),
- I_{CPU}, the energy consumed when the mobile device processes a query on its cache.
- $I_{network\_low}$, the energy consumed when the mobile device waits while the query is processed on the cloud.
- $I_{network\_high}$, 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, which can be computed using the formula, with *I* the current in Amps and *t* the processing time in hours:

$$Q(Ah) = I(A) \times t(h) \tag{10}$$

Therefore, we can simplify the computation of the energy consumption estimation by using only the electrical charge.

$$Q_{compute\_estimation} = I_{CPU} \times t_{compute\_estimation} \tag{11}$$

$$Q_{process\_mobile} = I_{CPU} \times t_{process\_mobile} \tag{12}$$

$$Q_{process\_cloud} = I_{network\_low} \times t_{exec\_query} + I_{network\_high} \times t_{send\_result} \tag{13}$$

$$Q_{total} = Q_{compute\_estimation} + Q_{process\_mobile} + Q_{process\_cloud} \tag{14}$$

This analysis and the background towards the different ways to use semantic caching and its relation to the cost estimations for the different steps of query processing in this environment are used in the proposed algorithm MOCCAD-Cache which is described in the next chapter, Chapter 3.

2.3 Cloud Query Processing

In addition to the query optimization on the mobile device, it is mandatory to take into account the optimization made on the cloud. As described in Chapter 1, several pricing models are proposed by different cloud providers for each of the cloud services they can sell. Also, several computation, transfer and storage models can be used regarding different cloud services. For example, Amazon Glacier [22] is a good solution in terms of monetary cost if most of the queries used on this storage service are insert operations and a few of them are read operations. Amazon S3 [23] is more relevant if the restriction on money is lower and if a quicker and more frequent access to the data is required.

Map Reduce is a programming model that permits distributed and parallel computing on a cluster [24]. Map Reduce is the heart of Hadoop, a framework made of a file system (HDFS) and a resource management platform (YARN). Several platforms for data management have been built on top of Hadoop, like Hive [25] or Pig [26].

A common way to represent queries on the cloud is to define them under directed acyclic graphs (DAGs) [27], [28]. Each node corresponds to operations and each edge represent the data flow. Each operation can be a logical operation in a query plan (operator graph), or a concrete operation to process the corresponding data (concrete operator graph). For those concrete operator graphs, the query execution and the cost estimation is similar to what has been explained in the previous section and in a more general way, equivalent to the RDBMS case. The difference between cloud query processing and query processing on a RDBMS is that the different possibilities to process a query are infinite on the cloud due to the infinite number of configurations and query plans. Also, query optimization on the cloud takes into consideration the money in addition of the time, making it a two dimensional optimization problem. To solve this problem, [28] first minimizes the processing time with a budget constraint, then minimizes the monetary cost with a time constraint and finally finds the trade-off between the suggested solutions. The authors use a greedy algorithm to assign each concrete operation to an operator respecting the constraint and then build the schedule for query processing. This algorithm is executed several times to generate several schedules. A filter operation will then choose the best of those schedules corresponding to the parameter to optimize and the given constraints. Given a number of operators with which the schedule can be built, the main steps to build this schedule are the following:

1. Get an operator to schedule.

2. Find operators ready for allocation (i.e. operators that are not dependent on other operators).
3. Gather all the operators that respect the constraints.
4. Choose the operator to add to the schedule based on the criteria such as balance container utilization, minimized network traffic, minimized completion time, etc…
5. Remove the operator from the list of ready operators.
6. Add the new ready operators to the ready list (i.e. the operators that were dependent on the chosen operator).
7. Insert the new operator into the schedule.
8. If there are still operators to schedule, go to step 1. Otherwise, start the Schedule Duration Estimation algorithm which will determine how much time and how much money the schedule costs.

After several iterations of this algorithm, it will be possible to determine the best schedule for the computed estimation. Even though estimates are not accurate, the chosen plan will be either the least expensive or close to the least expensive one, for the given constraints [20].

2.4 Conclusions

This chapter defines the ins and outs of semantic caching and shows the different works that can be used in order to integrate them with our algorithm. [14] defines very detailed and efficient algorithms for query trimming, which allow us to understand the different steps of semantic caching associated with the formal definitions of this structure given by [6] and [11]. With those definitions we can better understand the limitation of this architecture and algorithm, mainly in the cases of cache extended hit and partial hit. Indeed, in some conditions, there might be too much computation to process the query on the mobile device which can cost time and energy. Thus processing the query on the cloud may reduce this overhead. However, we still need to take into account the user constraints, especially on money, since processing queries on the cloud can be expensive. In addition, we explained the notions of implication and satisfiability given by [18] and used in the algorithm defined in [14] which are essential to understand how each input query can be compared with the queries in the query cache in order to know if a cache hit, a cache partial hit, a cache extended hit or a cache miss occurred. This chapter also presents the knowledge necessary to understand how to optimize queries in a general case, then on the cloud in order to pinpoint where optimizations need to be made and to understand the global optimization process in the mobile-cloud architecture. However, the way to estimate the cost to process a query on the mobile device still need to be determined.

Chapter 3: Proposed Solution and Architecture

In this chapter, we present our architecture and algorithm, called MOCCAD-Cache, to process queries while respecting user constraints in terms of time, energy and money. This algorithm contains the management of the different estimation computations and estimation caches, in addition to the already existing concepts of semantic caching presented in Chapter 2.

3.1 Cost Estimation on the Mobile Device

Chapter 2 presented the existing work on the cost estimation related to the process of a query on the cloud. This subsection now aims to describe the cost estimation related to the mobile device.

3.1.1 Cache Analysis Cost Estimation

The execution of the query trimming algorithm presented in Section 2.1, can be expensive. Indeed, we realize that the execution time and the energy consumption are already impacted during this cache analysis step. Therefore, regarding the different information items provided by the input query, it is possible to determine if there is a sufficient amount of energy and if it respects a minimum amount of time necessary for the cache analysis.

The idea for the energy and time estimation during the cache analysis is to compute statistics at the first launch of the application. Those statistics contain the time and the energy spent regarding the number of inequalities within the predicates of each cache entry. Let t_i be the necessary amount of time to analyze the semantic region *i* in the cache and e_i be the necessary amount of energy to analyze the semantic region *i*. Let n be the number of segments in the cache. The minimum amount of time necessary for the cache analysis is:

$$t_{min} = \sum_{i=1}^{n} t_i \tag{15}$$

Indeed, the worst case corresponds to the cache miss, where all the semantic regions are analyzed without finding any result. Thus, this is the minimum amount of time under which the user cannot process a query.

Identically, for the energy we have:

$$e_{min} = \sum_{i=1}^{n} e_i \tag{16}$$

3.1.2 Query Processing Cost Estimation on the Mobile Device

It is necessary to determine the time and the energy consumption needed for the query to be processed on the mobile device. This estimation depends mainly on the type of event that occurred during query trimming. Indeed, if a cache hit occurs, we consider that retrieving the query result is negligible in terms of time and energy consumption. Also, in the case of a partial cache hit, the time and energy to process the query is considered negligible in this first version of our algorithm. Indeed, certain types of semantic cache like the one from [14] carry on analyzing the cache with the remainder query. It is possible in this case to have several cache entries to evaluate while processing the probe query, which would consequently increase the processing cost. In the case of a cache miss, the time and the energy necessary for the query analysis has been consumed, and it is mandatory to process the query on the cloud. Therefore, no cost is taken into account to process the query on the mobile device. Finally, the cache extended hit can be a little bit more complex. In the case where the input query is equivalent to the one from the cache, the time and the energy necessary to load the query in the cache have mainly been consumed during the analysis of the cache. Evaluating the query to retrieve the result from the given cache region is considered negligible. However, if the input query implies the semantic region's query, then the query result is included inside the semantic region' set of tuples. It is therefore necessary to evaluate the query on each of the tuples contained in this set. The complexity of this type of evaluation depends mainly on the data structure used to store the tuples within the cache [19], as well as on the presence or not of indexes for the attributes that belong to the input query predicates [20]. In this first version of the 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 complexity of this operation *T(n)*, with n being the number of tuples and $T_{analyze\_tuple}$ the number of operations to analyze a tuple, is:

$$T(n) = n \times T_{analyze\_tuple} \tag{17}$$

$T_{analyze\_\text{tuple}}$ can be considered as the number of operations to assign each predicate attribute with the corresponding tuple attribute in order to check the satisfiability of the predicate with the given values. The time corresponding to those operations can be determined statistically by processing queries on several segments of different sizes.

3.1.3 Query Processing Cost Estimation on the Cloud

To estimate the execution time, the energy consumption and the monetary cost to process a query on the cloud, several possibilities worth considering.

First, it is possible to compute the estimation entirely on the mobile device. However, this method requires us to keep the metadata of the database hosted on the cloud updated as well as some information about the different nodes used for the query evaluation. This is therefore both complex due to the number of interactions to have with the cloud to retrieve those information items, and expensive in time due to the mobile device's low power resources. However the monetary cost relative to the computation of this estimation is negligible since it is done entirely on the mobile device.

Secondly, it is also possible to compute the estimation on the cloud. Indeed, we ask the cloud to compute the time and the amount of money needed to process the query on this platform. However, this requires us to use network connection to retrieve the estimation from the cloud, which uses more energy on the mobile device than the previous solution [7]. Also, the monetary cost will increase since the estimation is computed on the cloud servers. The advantage of such a solution is that the estimation is accurate and that additionally, it consumes less resource on the mobile device which would have been necessary for the estimation computation.

3.2 Design and Conception of the MOCCAD-Cache Algorithm

For the algorithm design, the organization of managers presented by [29] is used. Indeed, this approach divides the cache management into three independent parts (Figure 9):

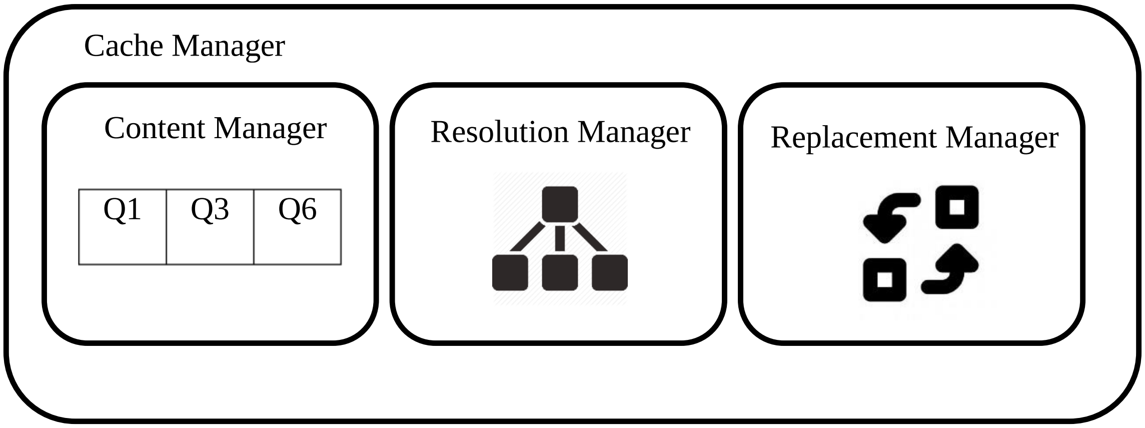


Figure 9 - Cache Manager Components

- 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 the replacement of data in the cache by some more relevant items. For example, if the used replacement policy is LRU (Least Recently Used), then the replacement manager will make sure to prepare the least recent items to be replaced by the new result. This situation only happens when the remaining cache space is not sufficient to insert the new query result without removing the old ones. In order to do this, it will store the dates of each entry or a linked list keeping every item in the right order.
- 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.

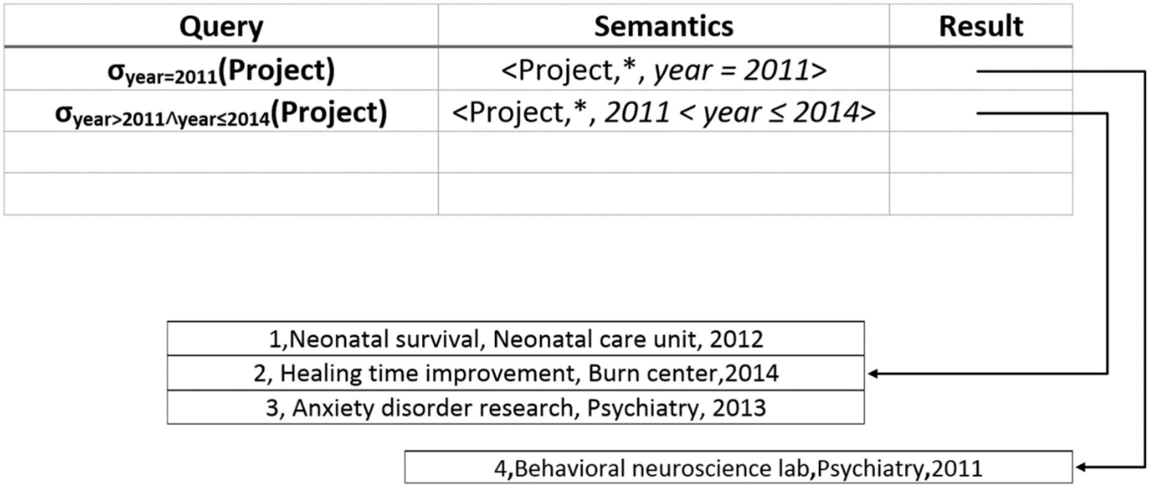
Even though each type of cache is a particular case, the organization with managers allows the important items involved in the algorithm to be modified properly so that it becomes easier to validate a possible solution for the given problem.

Also, three other types of managers are used in order to handle the computation of every estimation as well as the elaboration of a better solution. There exists a manager used to compute the execution time and the energy consumption while executing a query on the mobile device. This manager is called Mobile Estimation Computation Manager. There is also another manager used to compute the time, the mobile energy consumption and the monetary cost necessary to process a query on the cloud. This manager is called Cloud Estimation Computation Manager. Finally, another piece of the puzzle, the

Optimization Manager, needs to take care of the choice between the mobile query processing estimation and the cloud query processing estimation while respecting the constraints given by the user. Those constraints can be the remaining battery percentage, the maximum amount of time and the money to be paid to the cloud service providers for the query evaluation. Those managers are more accurately described in Section 3.2.2.

Three cache structures, query cache, mobile estimation cache, and cloud estimation cache, need to be used in order to be able to provide a solution for the given problem. However, each cache owns its content manager, replacement manager and resolution manager.

Query Cache: This cache is used to store the results of the previously executed queries. The type of cache used for this one is Semantic Caching (Figure 10).



Query	Semantics	Result
$\sigma_{year=2011}$(Project)	<Project,*, *year = 2011*>	
$\sigma_{year>2011 \wedge year \leq 2014}$(Project)	<Project,*, *2011 < year ≤ 2014*>	

1,Neonatal survival, Neonatal care unit, 2012
2, Healing time improvement, Burn center,2014
3, Anxiety disorder research, Psychiatry, 2013

4,Behavioral neuroscience lab,Psychiatry,2011

Figure 10 - Query Cache Example

Mobile Estimation Cache: This cache contains all the estimations corresponding to the query eventually processed on the mobile device. More specifically, it contains the execution time and the energy consumption for the query result retrieval within the query cache. The following figure shows an example of this estimation cache (Figure 11).

Query	Mobile Estimation
$\sigma_{age=11}$(Doctor)	{1.284s, 0.021mAh}
$\sigma_{year>2011 \land year\leq 2014}$ (Project)	{1.215s, 0.022mAh}

Figure 11 - Mobile Estimation Cache Example

Cloud Estimation Cache: This cache contains all the estimations corresponding to the query if it is processed on the cloud. Even though an estimation belongs to this cache, it does not mean that the query has been processed yet. The following figure shows an example of this estimation cache (Figure 12).

For both estimation caches, only the events of cache hit and cache miss can occur.

Query	Cloud Estimation
$\sigma_{age=20}$(Doctor)	{1.463s, 0.283mAh, $0.001}
$\sigma_{year=2011}$(Project)	{1.535s, 0.290mAh, $0.002}

Figure 12 - Cloud Estimation Cache Example

In order to design an algorithm that can decide whether a query should be run on the mobile device or on the cloud while consuming a minimum amount of time, energy and money, it is essential to define all components involved in the problem resolution process and to know how they interact with each other.

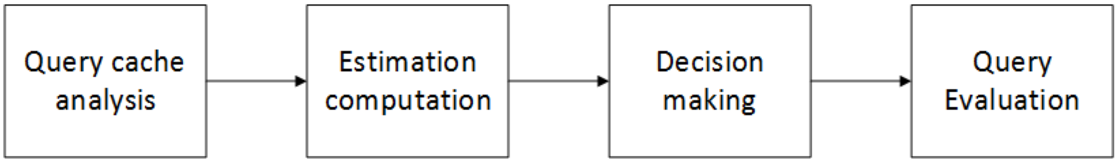


Figure 13 - The four main steps of the query processing algorithm

The MOCCAD-Cache algorithm consists of 4 main steps (Figure 13). 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 as well as the 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 evaluate the query. To describe the different activities happening in each of those steps, the following activity diagram is presented (Figure 14).

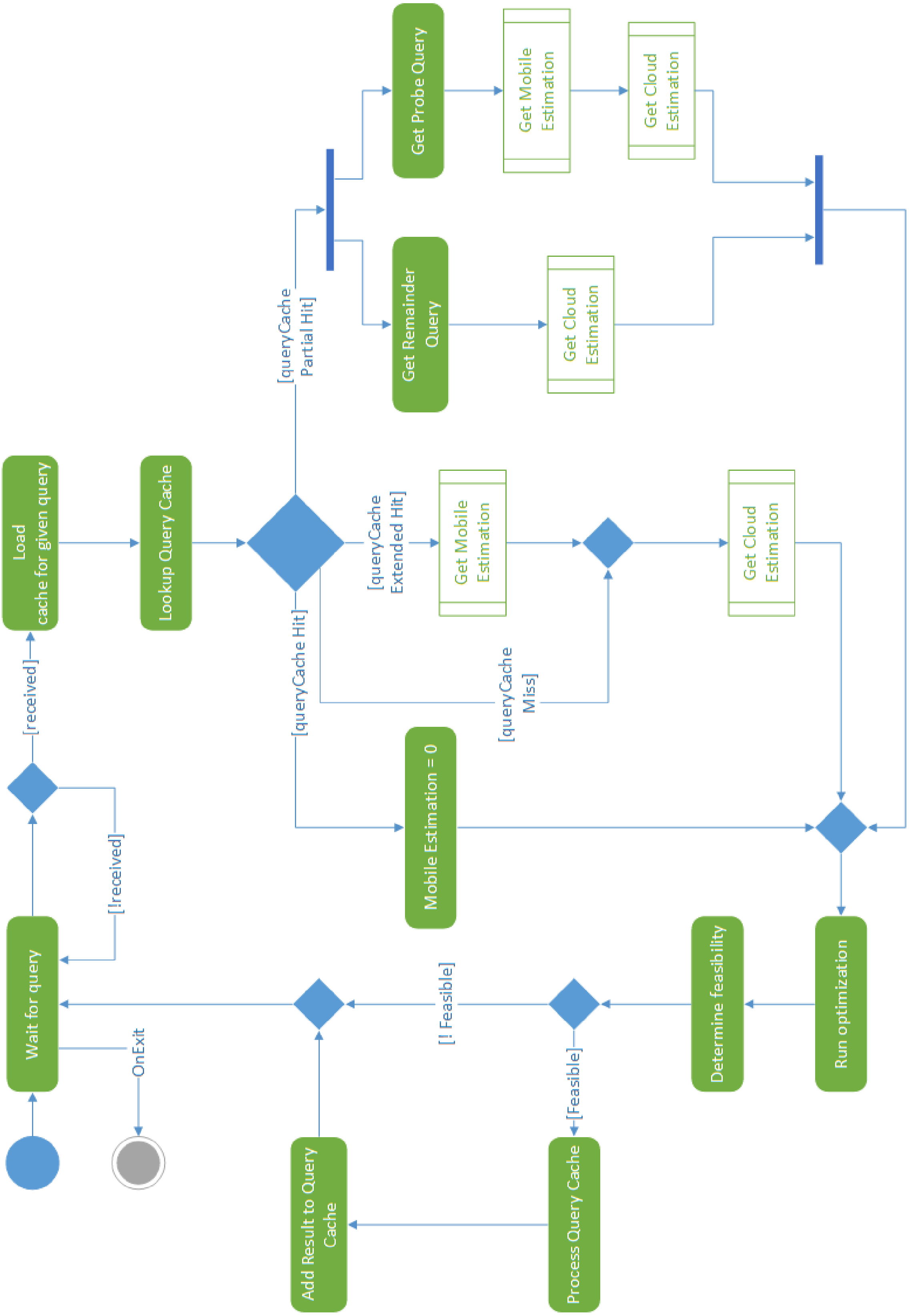


Figure 14 - Activity Diagram MOCCAD-Cache Algorithm

The two sub-processes, *Get Mobile Estimation* (Figure 15) and *Get Cloud Estimation* (Figure 16), describe the process to compute the estimation and store it in the estimation cache if it is not already contained in it.

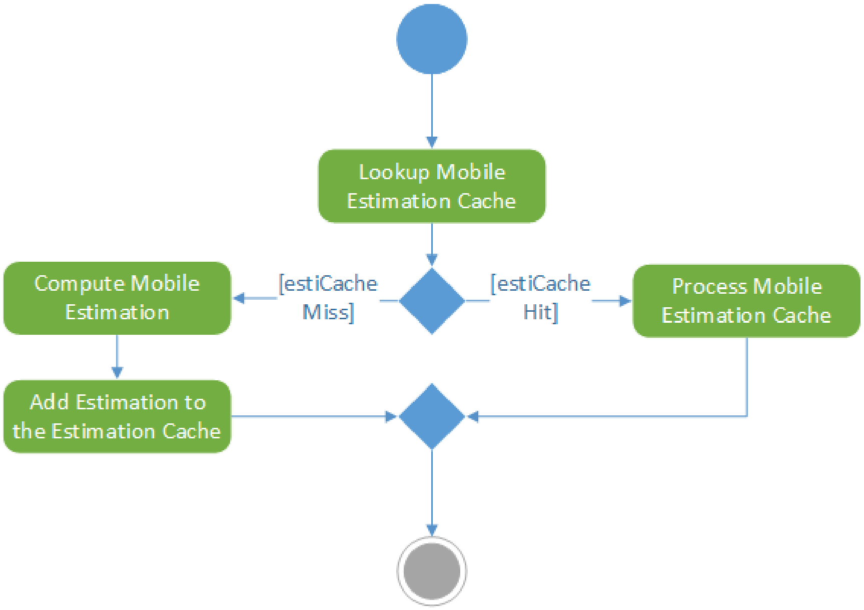

Lookup Mobile Estimation Cache
Compute Mobile Estimation
[estiCache Miss]
[estiCache Hit]
Process Mobile Estimation Cache
Add Estimation to the Estimation Cache

Figure 15 - Get Mobile Estimation Sub-Process

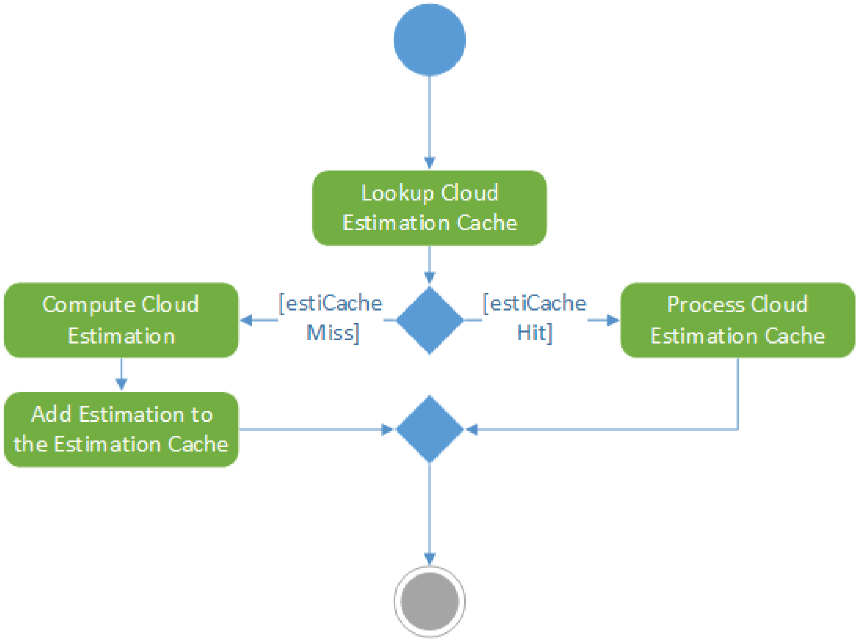

Lookup Cloud Estimation Cache
Compute Cloud Estimation
[estiCache Miss]
[estiCache Hit]
Process Cloud Estimation Cache
Add Estimation to the Estimation Cache

Figure 16 - Get Cloud Estimation Sub-Process

The following Figure 16 and Figure 17 describe those 4 different steps in details.

```
Algorithm 1 : CacheManager::load
Input: Query inputQuery, QueryCache queryCache, MobileEstimationCache mobileEstiCache, CloudEstimationCache cloudEstiCache,
MobileEstimationComputationManager mobileECompM,
CloudEstimationComputationManager cloudECompM, OptimizationManager oM.
Output: Query Result.
    mobileEstiResult ← ∞
    cloudEstiResult ← ∞

    queryLookup, pQuery, rQuery ← queryCache.lookup(inputQuery)
    if queryLookup = CACHE_HIT then
        mobileEstiResult ← 0

    else if queryLookup = PARTIAL_HIT then
        estiRemainder ← acquireEstimation( rQuery, cloudEstiCache, cloudECompM)
        estiProbe ← acquireEstimation( pQuery, mobileEstiCache, mobileECompM)
        mobileEstiResult ← estiRemainder + estiProbe
        cloudEstiCache ← acquireEstimation( inputQuery, cloudEstiCache, cloudECompM)

    else if queryLookup = EXTENDED_HIT then
        mobileEstiResult ← acquireEstimation( inputQuery, mobileEstiCache, mobileECompM)
        cloudEstiResult ← acquireEstimation( inputQuery, cloudEstiCache, cloudECompM)

    else if queryLookup = CACHE_MISS then
        cloudEstiResult ← acquireEstimation( inputQuery, cloudEstiCache, cloudECompM)
    end if

    bestEstimation, queryPlan ← oM.optimize(queryLookup, mobileEstiResult, cloudEstiResult)
    if bestEstimation != NIL then
        queryResult ← queryCache.process(queryPlan)
        queryCache.replace(queryResult, queryPlan)
    end if

```

Figure 17 – MOCCAD-Cache Algorithm

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

Figure 18 - Acquire Estimation Function

3.2.1 Query Evaluation Preparation

First of all, in order to estimate a query processing cost, it is necessary to determine how much time and how much energy will be spent to analyze the query cache. During the cache analysis phase, [18] provides its algorithm complexity needed to know if the posed query is satisfiable, if the posed query is equivalent to the query in the cache, if the posed query implies the query in the cache, or if the query in the cache implies the posed query. 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 the amount of 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. Indeed, since it depends on the computation resources specific to the mobile device, we do not need to compute it several times. 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 will need some additional pre-processed data. Indeed, even though we have all the information necessary to estimate the query processing cost on the mobile device, 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: relation name, number of tuples, average tuple size, maximum tuple size, attribute names, the minimum value for each attribute, the maximum value for each attribute, and the number of different values for each attribute. Those information items are downloaded the first time when the algorithm accesses the database. We assume that the database on the cloud is never altered.

3.2.2 Inputs

This algorithm first requires the query to be processed as an input. This query, as we have previously described it, is a selection query (without any projection attributes) on one relation (no joins). Secondly, this algorithm needs the three different caches: the query cache, the mobile estimation cache and the cloud estimation cache. Finally it is necessary to provide the different managers responsible for computing the estimations as follows:

Mobile Estimation Computation Manager: it handles the computation of the estimation corresponding to the query execution on the mobile device. The goal of this manager is therefore to estimate as accurately as possible the execution time and the energy consumption to evaluate a query on the query cache. In the case of an exact hit, these consumed time and energy are considered negligible. Indeed, there is not any computation to do to retrieve the query result. Also, in the case of a cache miss, none of the tuples in the cache needs to be retrieved. The execution time and the energy consumption are thus considered negligible as well. When a partial hit occurs, the result contained in the usable cache entry can be retrieved without any computation. Finally, in the case of an extended hit, there are two possibilities: either the query corresponding to the semantic region is equivalent to the posed query or the posed query result is included inside the semantic region. In the case where both queries are equivalent, the query evaluation is similar to the cache hit case. The time and the energy in those cases have already been consumed by the cache analysis. However when the posed query result is contained in a semantic region, each tuple will be evaluated to see if it satisfies the input query predicates. At this point the execution time and energy consumption estimation are involved. The third extended hit case is not handled as explained in the previous section.

Cloud Estimation Computation Manager: it handles the computation of the estimation corresponding to the query processing on the cloud. It asks the cloud to estimate the necessary amount of time and amount of money to process the query on its servers. Thanks to the downloaded database metadata, it is possible to estimate the size of the result retrieved from the cloud. Consequently, it is possible to determine the execution time of this query thanks to the used network bandwidth. Regarding the energy consumed while using the network, several energy states have been defined in Section 2.2.3, allowing the proposed algorithm to determine the energy consumption from executing the query on the cloud and retrieving the query result.

Optimization Manager: the optimization manager is responsible for defining the query plan from the different estimations computed beforehand and making sure that it respects the user-specified constraints.

3.2.3 Query Cache Analysis

First of all, we analyze the query cache (Line 4 in Figure 17). The query cache calls its content manager which will return 3 different items:

- The type of cache hit or miss.
- The probe query corresponding to the query is used to retrieve data from the cache in the case of a partial hit (*PARTIAL_HIT*). This probe query is null otherwise.
- The remainder query corresponding to the query is used to retrieve data from the cloud in the case of a partial hit (*PARTIAL_HIT*). This remainder query is null if a partial hit did not occur.

3.2.4 Estimation Regarding the Query Cache Analysis Result

This subsection describes the different steps of the algorithm from Line 5 to Line 20 in Figure 17.

In the case of a query cache 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 cost to process the probe query on the mobile device as well as the cost to process the remainder query on the cloud. Those two estimation 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 function define in Figure 18. This function looks for those estimations in the cloud or mobile estimation caches (Figure 18 – Line 1). If they are not available in those caches, then the estimation is computed thanks to the estimation computation manager and the estimation cache calls its replacement manager to insert the new estimation into the cache (Figure 18 – Line 4 to Line 6). This way, even though the query is not executed, the estimation still belongs to the cache. This works the same way either for the cloud estimation cache or for 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.

3.2.5 Decision

This subsection describes the decision making part (Line 22 in Figure 17).

Once the estimations have been computed, it is now possible to make a decision of whether the query should be processed fully on the mobile device, the query should be processed fully on the cloud, or a part of the query should be processed on the mobile device and a part of the query should be processed on the cloud. A corresponding query plan (*queryPlan*) is then generated for the query. For instance, in the case of a partial hit, the optimization manager will compare the estimation to process the probe query on the mobile device and the remainder query on the cloud with the estimation to process the input query entirely on the cloud. One possibility is to ask for the cloud to return the estimations for several possible configurations (i.e. 1, 10, 50, 100 small, medium or big instances). This, however, would increase the monetary cost related to the estimations on the cloud. Thus, we consider that the cloud returns its estimation for the given constraints that have been given to it, and the mobile device does not need to make any choice upon that.

3.2.6 Query Evaluation

From Line 23 to Line 26 in Figure 17, the different steps correspond to the query evaluation part.

If it is possible to process the query while respecting the constraints in terms of execution time, energy consumption and monetary cost, then the query cache (*QueryCache*) calls its resolution manager to process the generated query plan and return the result. 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 in 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 than the first one to retrieve the result, then the estimation is not accurate anymore. However, a possible solution could be to store this estimation elsewhere for replacement purposes. Finally, after the replacement, the query result is sent to the user.

3.3 Algorithm Running Illustration

In order to illustrate the running of this algorithm, we present, in this section, several examples of the cache behavior in the different possible cases (cache hit, partial hit, extended hit and cache miss). These examples are based on the following entity-relationship diagram (Figure 19) that represents the organization of a database storing projects involving doctors and patients.

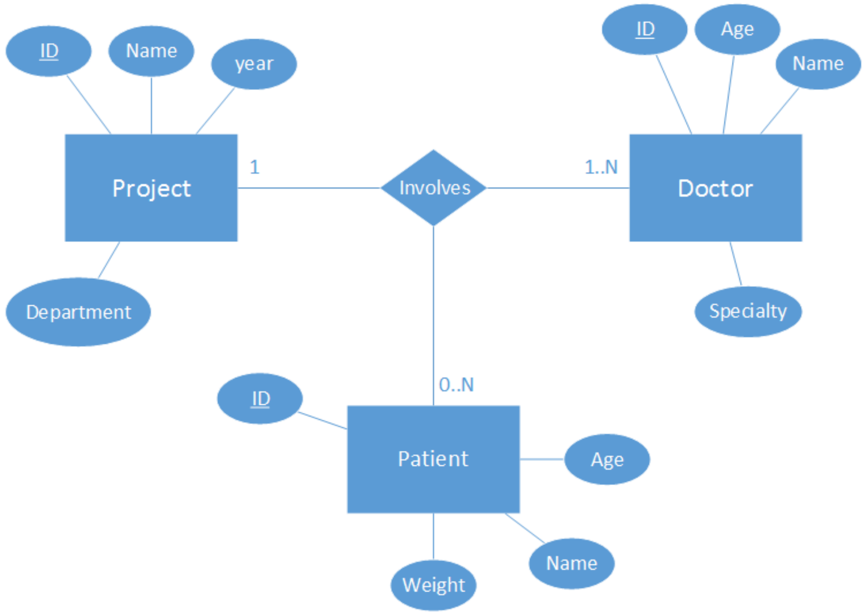


Figure 19 - Entity-Relationship Diagram for Algorithm Illustration

We also provide some dummy content for the table *Patient*, *Doctor* and *Project* (Figures 20, 21 and 22):

Project ID	Project Name	Project Department	Project year
001	Neonatal survival	Neonatal care unit	2012
002	Healing time improvement	Burn center	2014
003	Anxiety disorder research	Psychiatry	2013
004	Behavioral neuroscience lab	Psychiatry	2011

Figure 20 - ***Project*** **Table**

Doctor ID	Doctor Name	Doctor Age	Doctor Specialty
001	Bert Wade	32	Psychiatry
002	Dustin Rhodes	45	Psychiatry
003	Joy Rodriguez	35	Surgery
004	Jonathan Garrett	48	Neurological Surgery
005	Lela Holloway	52	Pediatric Medicine

Figure 21 - *Doctor* Table

Patient ID	Patient Name	Patient Age	Patient Weight (kg)
001	June Howard	25	85.2
002	Maurice Myers	23	72.8
003	Wilbert Stewart	45	95.6
004	Nora Baldwin	1	10.3
005	Jody Gregory	12	55.0
006	Georges Abidbol	42	72.3

Figure 22 - *Patient* Table

In order to simplify each example, only the primary keys will be written for the tuples contained in the query cache. Also the semantic region is presented in its simplest version. The chosen replacement policy will be LRU. All the following examples are presented in the chronological order. The estimation values are arbitrary. The user specifies the following constraints per queries:

$$time \leq 1.5s$$

$$money \leq \$0.003$$

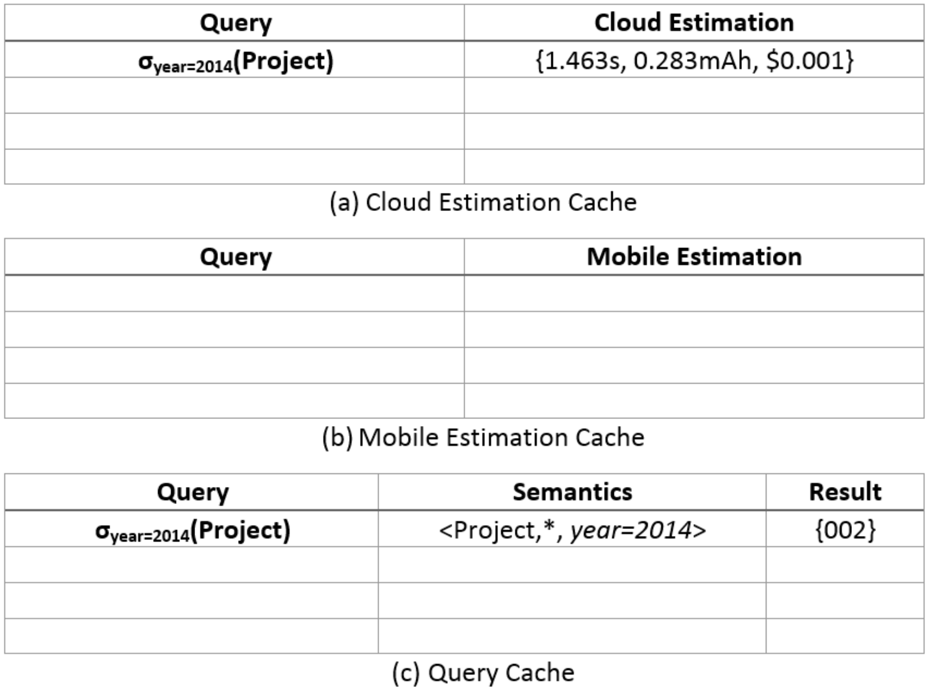
$$energy \leq 0.320mAh$$

3.3.1 Cache Hit

The input query is the following:

$$\sigma_{year=2014}(PROJECT)$$

The objective parameter chosen by the user is time. The cache content before and after the query execution is presented in Figure 23. When we look into the query cache, we find exactly the same query. In this case, the execution time, the energy consumption and the monetary cost are considered negligible. The decision is therefore easy to make and the query is executed on the mobile device, that is to say, on the query cache. The cache does not change since the retrieved query result is already contained in the query cache.



Query	Cloud Estimation
$\sigma_{year=2014}$(Project)	{1.463s, 0.283mAh, $0.001}

(a) Cloud Estimation Cache

Query	Mobile Estimation

(b) Mobile Estimation Cache

Query	Semantics	Result
$\sigma_{year=2014}$(Project)	<Project,*, *year=2014*>	{002}

(c) Query Cache

Figure 23 - Caches' States before and after Cache Hit Example

3.3.2 Cache Partial Hit

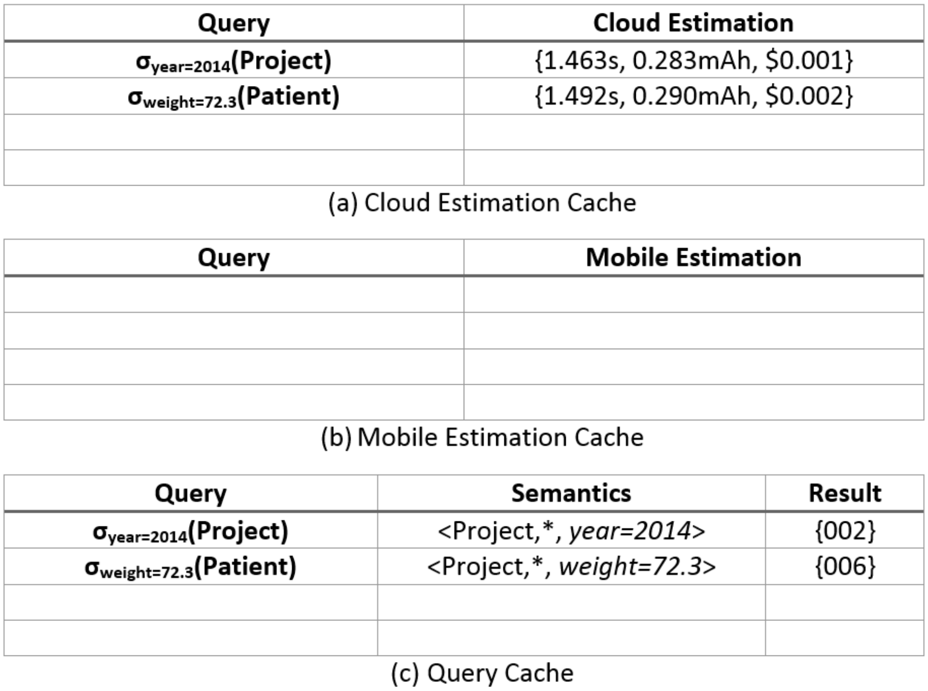
The input query is the following:

$$\sigma_{weight \leq 72.3}(PATIENT)$$

The objective parameter to minimize, which is specified by the user, is the time. The content of the cache before the query execution is presented in Figure 24. In this example, we assume that a query has been posed in order to retrieve all the patients with a weight of 72.3 kg. The cost estimation to process this query on the cloud is now in the cloud estimation cache (Figure 24.a) and its result is in the query cache (Figure 24.c). When we look for the input query in the cache, *weight ≤ 72.3* does not imply *weight = 72.3*. However, *weight ≤ 72.3 ∧ weight = 72.3* is satisfiable [18]. Indeed, there exists at least one value that matches the conjunction of those two inequalities (i.e. a weight of 72.3 kg). Therefore, this corresponds to a partial hit, and two sub-queries are created thanks to the algorithm presented in [14]. The probe query ($\sigma_{weight \leq 72.3}(PATIENT)$) would retrieve the patient(s) in the cache entry corresponding to $\sigma_{weight = 72.3}(PATIENT)$ on the mobile device, and the remainder query ($\sigma_{weight < 72.3}(PATIENT)$) would retrieve the right patient(s) from the cloud. Here the remainder query is simplified for illustration purposes. The remainder query as it is generated by the algorithm presented in [14] would be represented as $\sigma_{weight \leq 72.3 \wedge \neg (weight = 72.3)}\ (PATIENT)$.

The probe query estimation is computed because it is not contained in the cache. This estimation is then stored in the mobile estimation cache. The remainder query estimation is also computed and stored in the cloud estimation cache for the same reason. The whole query processing estimation to retrieve the result with the probe query and the remainder query is equal to the sum of those two acquired estimations. This solution does not handle parallelism to process the probe query on the mobile device at the same time that it executes the remainder query on the cloud. The values for the probe query estimation are the time of 0.480s and the energy consumption of 0.030mAh. The values for the remainder query estimations are the time of 1.496s, the energy consumption of

0.308mAh and the monetary cost of $0.003 Therefore the estimation tuple to process the whole query is {1.976s, 0.338 mAh, $0.003}.



Query	Cloud Estimation
$\sigma_{year=2014}$(Project)	{1.463s, 0.283mAh, $0.001}
$\sigma_{weight=72.3}$(Patient)	{1.492s, 0.290mAh, $0.002}

(a) Cloud Estimation Cache

Query	Mobile Estimation

(b) Mobile Estimation Cache

Query	Semantics	Result
$\sigma_{year=2014}$(Project)	<Project,*, *year=2014*>	{002}
$\sigma_{weight=72.3}$(Patient)	<Project,*, *weight=72.3*>	{006}

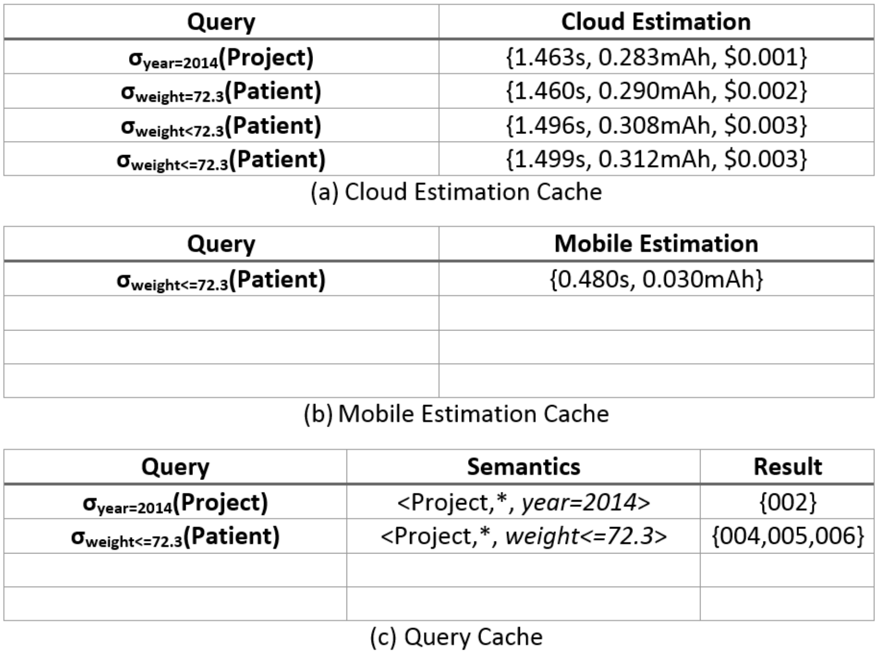
(c) Query Cache

Figure 24 - Caches' States before the Partial Hit Example

The second possibility is to process the whole input query directly on the cloud. Since the input query $\sigma_{weight\leq 72.3}(PATIENT)$ is not contained in the cache, we also need to compute the estimation and to store it in the cloud estimation cache. The estimated costs to process this query are 1.499s, 0.312 mAh and $0.003.

Concerning the estimations to process a query on the cloud, we first retrieve the time, energy and money used by its services to process this query. Also, we determine the estimated transfer time thanks to the Formulas 5 to 9 in in Chapter 2. Finally for the cloud estimation, we determine the amount of energy that would be spent on the mobile device with the Formulas 11 to 14 in Chapter 2.

The optimization manager will then build a query plan. Here, we can see that the estimation involving query processing on the mobile does not respect the constraint of energy consumption. Therefore, since running straightly the query on the cloud respects all the constraints, the optimization manager chooses to build the query plan to process such a query. The result is then presented to the user. Finally the query result is stored into the cache and the caches' content becomes the following (Figure 25).



Query	Cloud Estimation
$\sigma_{year=2014}$(Project)	{1.463s, 0.283mAh, \$0.001}
$\sigma_{weight=72.3}$(Patient)	{1.460s, 0.290mAh, \$0.002}
$\sigma_{weight<72.3}$(Patient)	{1.496s, 0.308mAh, \$0.003}
$\sigma_{weight<=72.3}$(Patient)	{1.499s, 0.312mAh, \$0.003}

(a) Cloud Estimation Cache

Query	Mobile Estimation
$\sigma_{weight<=72.3}$(Patient)	{0.480s, 0.030mAh}

(b) Mobile Estimation Cache

Query	Semantics	Result
$\sigma_{year=2014}$(Project)	<Project,*, *year=2014*>	{002}
$\sigma_{weight<=72.3}$(Patient)	<Project,*, *weight<=72.3*>	{004,005,006}

(c) Query Cache

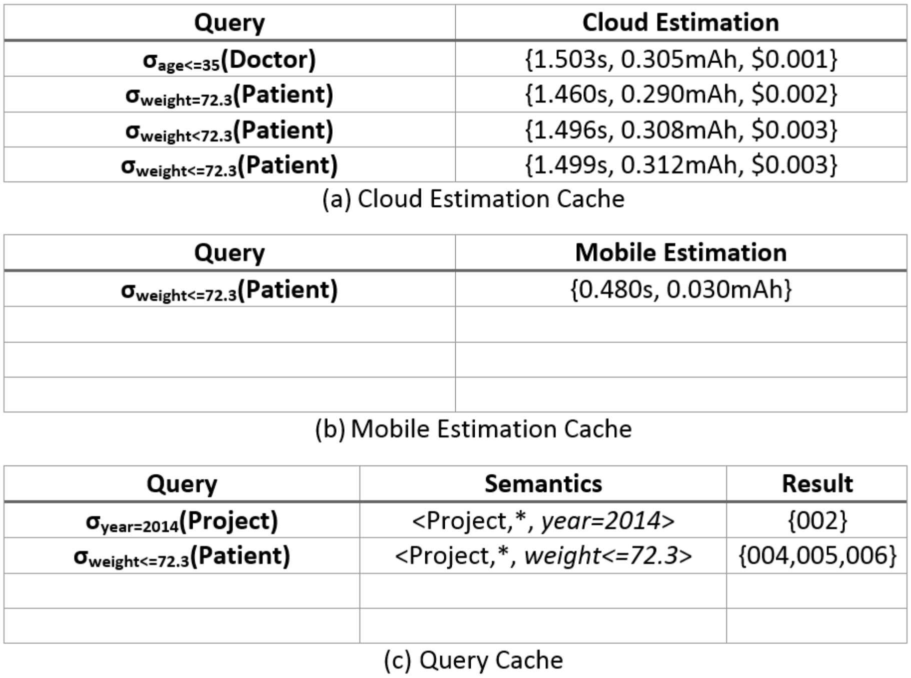
Figure 25 - Caches' States after the Partial Hit Example and before the Cache Miss Example

3.3.3 Cache Miss

The input query is the following:

$$\sigma_{age\leq 35}(DOCTOR)$$

The content of the cache before the query execution is presented Figure 25. In the current case, none of the entries in cache can be used to answer to the input query. Hence, we look for the estimation for the input query inside the cloud estimation cache. Since there is none, the estimation is therefore computed. The result is the following estimation tuple: {1.503s, 0.305mAh, $0.001}. At that moment, we want to add this estimation to the cloud estimation cache, but this one is full. We thus replace the oldest estimation in the cache, assuming that it is using the LRU replacement policy. Then, the optimization manager fails to generate any query plan since the time estimation does not respect the time constraint specified by the user. Thus, the query is not processed and nothing is added to the cache. Figure 26 shows the caches' states after the execution of this query and before the next one.



Query	Cloud Estimation
$\sigma_{age<=35}$(Doctor)	{1.503s, 0.305mAh, $0.001}
$\sigma_{weight=72.3}$(Patient)	{1.460s, 0.290mAh, $0.002}
$\sigma_{weight<72.3}$(Patient)	{1.496s, 0.308mAh, $0.003}
$\sigma_{weight<=72.3}$(Patient)	{1.499s, 0.312mAh, $0.003}

(a) Cloud Estimation Cache

Query	Mobile Estimation
$\sigma_{weight<=72.3}$(Patient)	{0.480s, 0.030mAh}

(b) Mobile Estimation Cache

Query	Semantics	Result
$\sigma_{year=2014}$(Project)	<Project,*, *year=2014*>	{002}
$\sigma_{weight<=72.3}$(Patient)	<Project,*, *weight<=72.3*>	{004,005,006}

(c) Query Cache

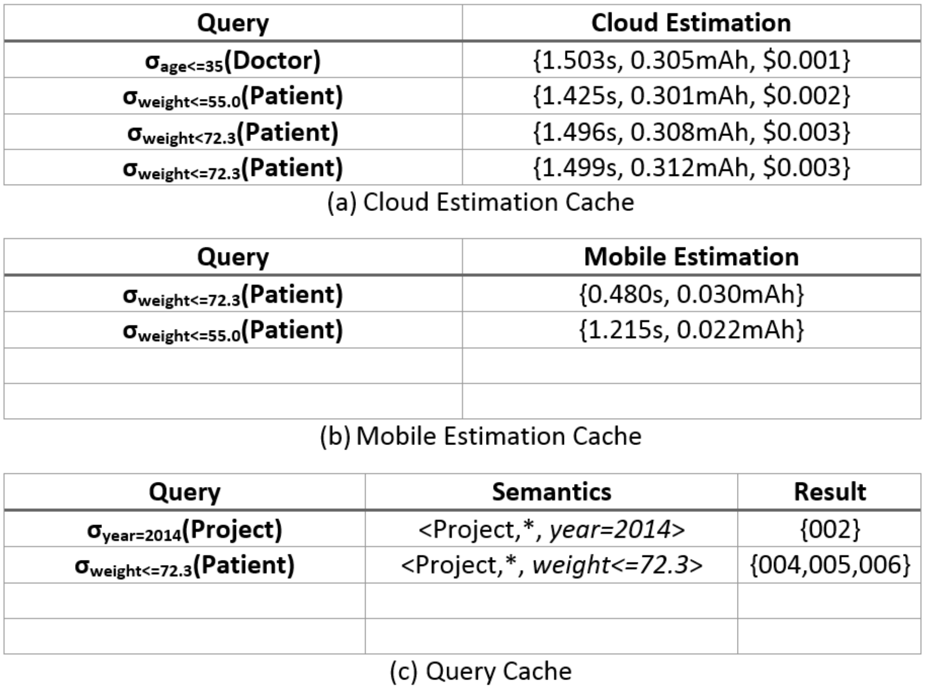
Figure 26 - Caches' States after the Cache Miss Example and before the Extended Hit Example

3.3.4 Cache Extended Hit

The input query is the following:

$$\sigma_{weight \leq 55.0}(DOCTOR)$$

The objective parameter to minimize is money. During the cache analysis part of the query processing algorithm for this input query, we can see that *weight* ≤ *55.0* implies *weight* ≤ *72.3*. This corresponds to an extended hit. However *weight* ≤ *72.3* does not imply *weight* ≤ *55.0*. Hence, the input query and the cache segment query are not equivalent, which means the result of the input query is included inside the corresponding semantic region. First of all, we need to compute the estimation to retrieve the tuples which match the input query predicates since this estimation is not contained in the mobile estimation cache. We use Formula 7 from Section 3.1.2 in Chapter 2 to compute the estimated execution time and Formula 15 from Section 2.2.3 in Chapter 2 to compute the estimated energy consumption. This estimation is stored in the mobile estimation cache. With the cloud estimation cache being full, we replace the estimation for the query ($\sigma_{weight=72.3}(PATIENT)$) with the estimation for the query ($\sigma_{weight\leq 55.0}(PATIENT)$). Let's assume that the costs to process the query on the cloud are the following: 1.450s for the execution time, 0.300mAh for the energy consumption and $0.002 for the monetary cost. Let's assume that the costs to process the query on the mobile device are the following: 1.200s for the execution time and 0.020mAh for the energy consumption. Since the mobile device estimation is better than the cloud estimation in terms of monetary cost (the chosen objective parameter to minimize), and that it takes less than 1.5s, consumes less than 0.320mAh and cost less than $0.003, the optimization manager will create the query plan to process the query on the mobile device. Since the query has been processed on the mobile device, we do not add an entry in the cache to store the result because the tuples are already contained in another segment in the cache. The result is then given to the user. The content of the cache after this query has been processed is shown in Figure 27.



Query	Cloud Estimation
$\sigma_{age<=35}$(Doctor)	{1.503s, 0.305mAh, $0.001}
$\sigma_{weight<=55.0}$(Patient)	{1.425s, 0.301mAh, $0.002}
$\sigma_{weight<72.3}$(Patient)	{1.496s, 0.308mAh, $0.003}
$\sigma_{weight<=72.3}$(Patient)	{1.499s, 0.312mAh, $0.003}

(a) Cloud Estimation Cache

Query	Mobile Estimation
$\sigma_{weight<=72.3}$(Patient)	{0.480s, 0.030mAh}
$\sigma_{weight<=55.0}$(Patient)	{1.215s, 0.022mAh}

(b) Mobile Estimation Cache

Query	Semantics	Result
$\sigma_{year=2014}$(Project)	<Project,*, *year=2014*>	{002}
$\sigma_{weight<=72.3}$(Patient)	<Project,*, *weight<=72.3*>	{004,005,006}

(c) Query Cache

Figure 27 - Caches' States after the Extended Hit Example

3.4 Conclusions

In this chapter we described the proposed algorithm, MOCCAD-Cache, that can make a decision of whether or not a query should be run regarding the estimations and the user constraints, and also where, on the mobile device or on the cloud, it should be ran. In order to be able to maximize the number of queries which respect their user constraints, we want to use as much as possible the query cache to retrieve less data from the server and thus, reduce the amounts of time, energy and money to get the query result. This is why we use semantic caching in our architecture. However, in some cases like the cache extended hit, processing a query on the mobile device might be more expensive than processing the query on the cloud due to the number of tuples which reside in the cache, the time and energy to analyze those tuples and the low computing resources of the mobile device compared to the cloud. Therefore, we compute the cost estimations to make a decision of whether to run the query on the mobile device or on the cloud based on the user constraints. Those user constraints need to be respected, and the query processing estimations allow us to make sure of this before we launch the execution. However, computing an estimation requires time and adds an overhead to the query process, which led us to use the estimation caches in order to reduce this overhead over time.

Chapter 4: Experimentation and Results

This chapter describes the experiments evaluating the performance of the MOCCAD-Cache algorithm. It details the design and implementation of the prototype, and presents the experiment model, the generation of test data and finally the performance results.

4.1 Prototype

4.1.1 Environment

The MOCCAD-Cache algorithm as well as all the associated algorithms Guo et al. [18] and Ren et al. [14] have been implemented on Android [30]. Android is an open source operating system created by Google. It is based on a Linux kernel, and works on several mobile devices such as smart-phones and tablets. It can also be found in televisions, watches and cars. Android applications are developed in Java and use the virtual machine Dalvik, which allows such applications to be deployed on many devices. The Android current version is Android Lollipop 5.0 but the prototype created for this experimentation has been developed on Android Kit Kat 4.4.3. All the experiments have been run on a HTC One M7ul (Figure 28) embedding a Qualcomm Snapdragon 600 with a 1.728 GHz CPU, 2GB of RAM and a battery capacity of 2300 mAh.

Figure 28 - HTC One M7

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 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 thanks to HiveQL, It can also return the metadata items related to the stored relation(s) thanks to the Hive Metastore. This Hive Metastore uses a MySQL Server (Version 5.1.73) to store the metadata.

4.1.2 Project Organization

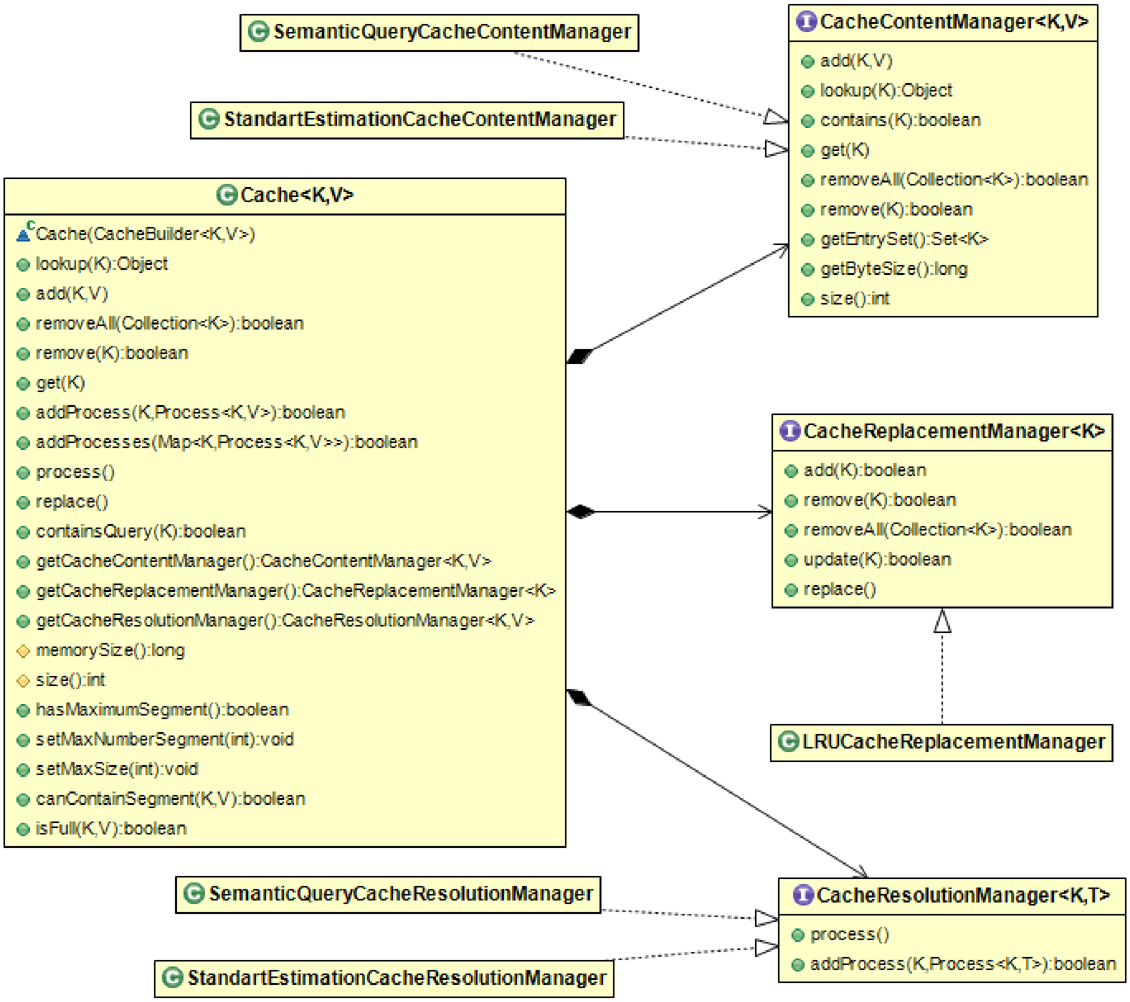
The implementation of the prototype on the mobile device is divided into 2 parts:

- *AndroidCachePrototype*: the Android application allowing the user to build the queries, specify constraints, and view the statistics on previously run queries.
- *CacheLibrary*: the library used by the Android application. It defines classes and methods allowing the application to load queries into a chosen

type of cache. The implementation of the MOCCAD-Cache algorithm is available in this library.

The AndroidCachePrototype subproject uses different classes and methods from the CacheLibrary project in order to use the caching data structures as well as estimate and optimize query processing.

4.1.3 Design and Implementation



SemanticQueryCacheContentManager
StandartEstimationCacheContentManager
CacheContentManager<K,V>
add(K,V)
lookup(K):Object
contains(K):boolean
get(K)
removeAll(Collection<K>):boolean
remove(K):boolean
getEntrySet():Set<K>
getByteSize():long
size():int
Cache<K,V>
Cache(CacheBuilder<K,V>)
lookup(K):Object
add(K,V)
removeAll(Collection<K>):boolean
remove(K):boolean
get(K)
addProcess(K,Process<K,V>):boolean
addProcesses(Map<K,Process<K,V>>):boolean
process()
replace()
containsQuery(K):boolean
getCacheContentManager():CacheContentManager<K,V>
getCacheReplacementManager():CacheReplacementManager<K>
getCacheResolutionManager():CacheResolutionManager<K,V>
memorySize():long
size():int
hasMaximumSegment():boolean
setMaxNumberSegment(int):void
setMaxSize(int):void
canContainSegment(K,V):boolean
isFull(K,V):boolean
CacheReplacementManager<K>
add(K):boolean
remove(K):boolean
removeAll(Collection<K>):boolean
update(K):boolean
replace()
LRUCacheReplacementManager
SemanticQueryCacheResolutionManager
StandartEstimationCacheResolutionManager
CacheResolutionManager<K,T>
process()
addProcess(K,Process<K,T>):boolean

Figure 29 - Cache Managers UML Class Diagram

Figure 29 shows the class diagram representing the content of a cache and its different cache managers. A Cache instance is created thanks to a Cache builder pattern allowing to specify different parameters and managers such as a CacheContentManager instance, a CacheReplacementManager instance, a CacheResolutionManager instance, the maximum number of segments and the maximum size of the cache. As shown on the diagram, several implementations of the different managers have been done and can be given to the Cache builder in order to build the Cache instance. The CacheContentManager class and the CacheResolutionManager class are inherited by two different managers and instantiated regarding the type of cache it uses. In the built system, a query cache and an estimation cache can be used and can be managed in different ways. Concerning the cache replacement manager, only one specialization is needed for now since the different caches only use the LRU replacement policy.

Figure 30 shows the possible query processors that can be used. A query processor owns a data access provider allowing it to connect with a server able to return some data. For example, in this experimentation, it is used to send queries to a Web Service (aka the data owner) in order to process the queries in a cloud.

The prototype provides the user with different features such as creating a new query, viewing the result tuples, viewing the different caches, viewing the already processed queries and modifying the settings (constraints, types of caches, experimentation options, etc…). To build a new query, the user can use the interface in Figure 31.a.

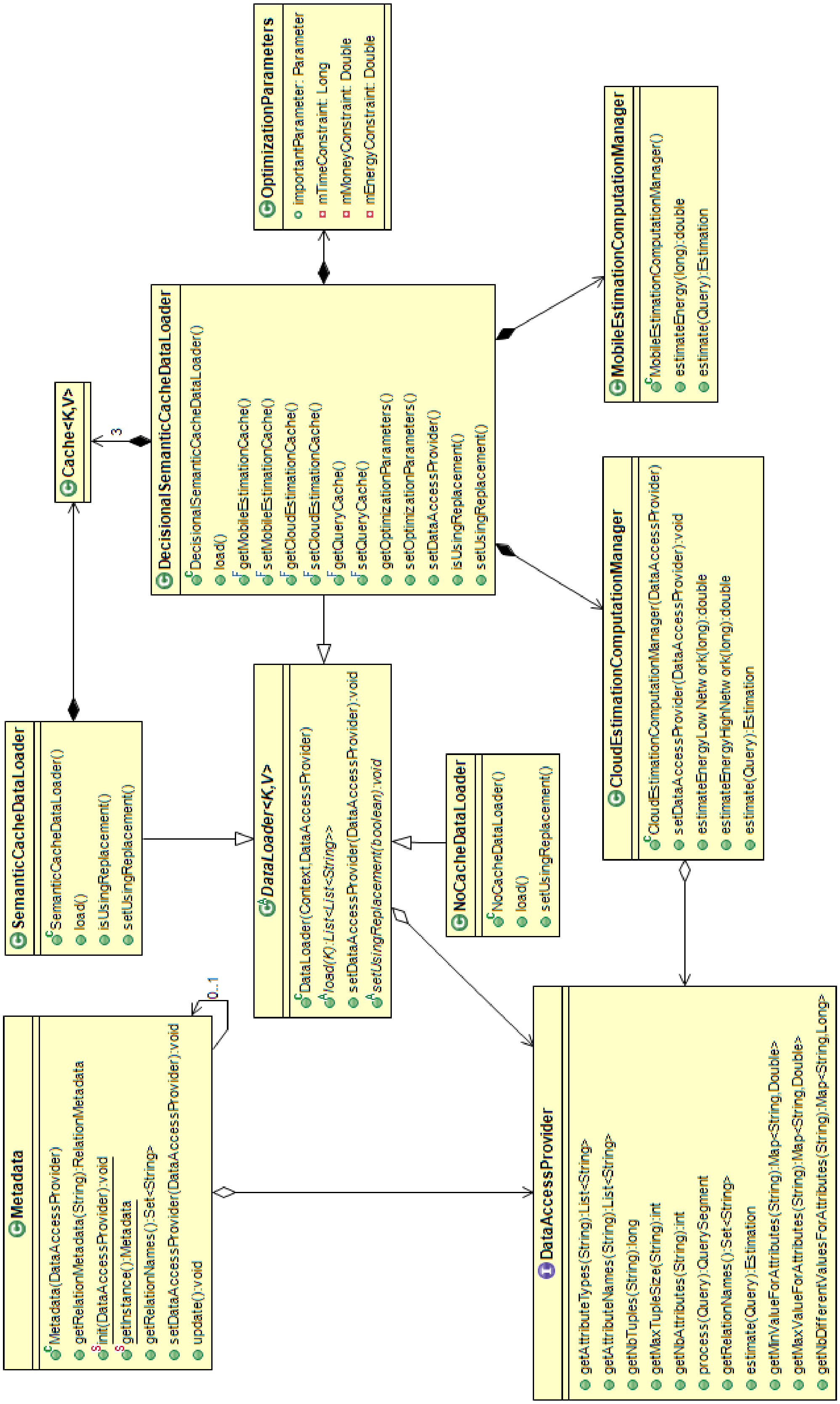
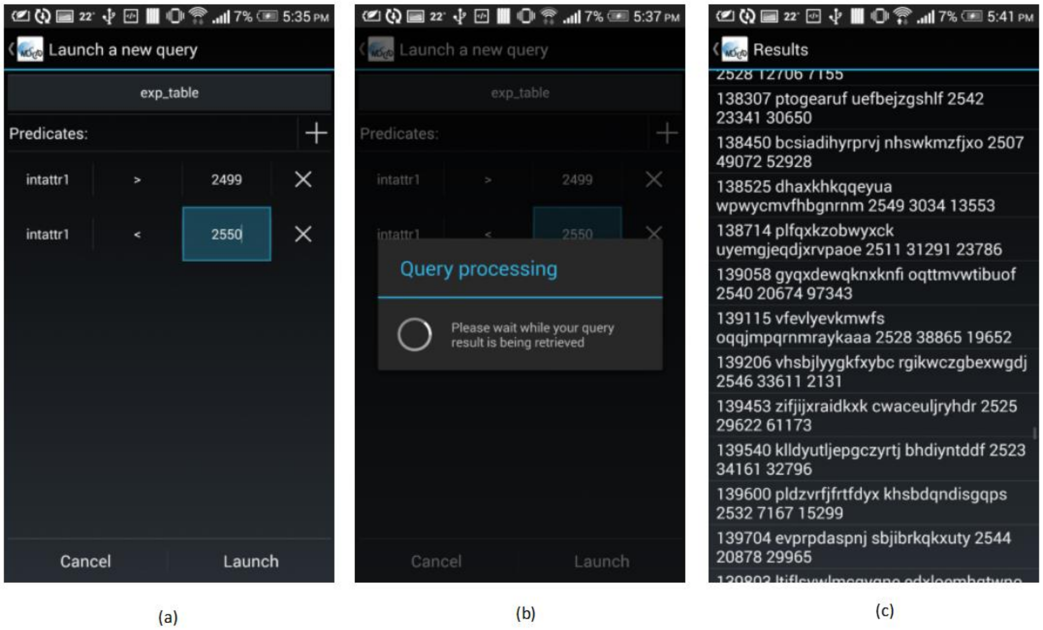

OptimizationParameters
importantParameter: Parameter
mTimeConstraint: Long
mMoneyConstraint: Double
mEnergyConstraint: Double
MobileEstimationComputationManager
MobileEstimationComputationManager()
estimateEnergy(long):double
estimate(Query):Estimation
Cache<K,V>
3
DecisionalSemanticCacheDataLoader
DecisionalSemanticCacheDataLoader()
load()
getMobileEstimationCache()
setMobileEstimationCache()
getCloudEstimationCache()
setCloudEstimationCache()
getQueryCache()
setQueryCache()
getOptimizationParameters()
setOptimizationParameters()
setDataAccessProvider()
isUsingReplacement()
setUsingReplacement()
CloudEstimationComputationManager
CloudEstimationComputationManager(DataAccessProvider)
setDataAccessProvider(DataAccessProvider):void
estimateEnergyLowNetwork(long):double
estimateEnergyHighNetwork(long):double
estimate(Query):Estimation
SemanticCacheDataLoader
SemanticCacheDataLoader()
load()
isUsingReplacement()
setUsingReplacement()
DataLoader<K,V>
DataLoader(Context,DataAccessProvider)
load(K):List<List<String>>
setDataAccessProvider(DataAccessProvider):void
setUsingReplacement(boolean):void
NoCacheDataLoader
NoCacheDataLoader()
load()
setUsingReplacement()
Metadata
Metadata(DataAccessProvider)
getRelationMetadata(String):RelationMetadata
init(DataAccessProvider):void
getInstance():Metadata
getRelationNames():Set<String>
setDataAccessProvider(DataAccessProvider):void
update():void
0..1
DataAccessProvider
getAttributeTypes(String):List<String>
getAttributeNames(String):List<String>
getNbTuples(String):long
getMaxTupleSize(String):int
getNbAttributes(String):int
process(Query):QuerySegment
getRelationNames():Set<String>
estimate(Query):Estimation
getMinValueForAttributes(String):Map<String,Double>
getMaxValueForAttributes(String):Map<String,Double>
getNbDifferentValuesForAttributes(String):Map<String,Long>

Figure 30 - Query Processors UML Class Diagram



exp_table

intattr1 > 2499

intattr1 < 2550

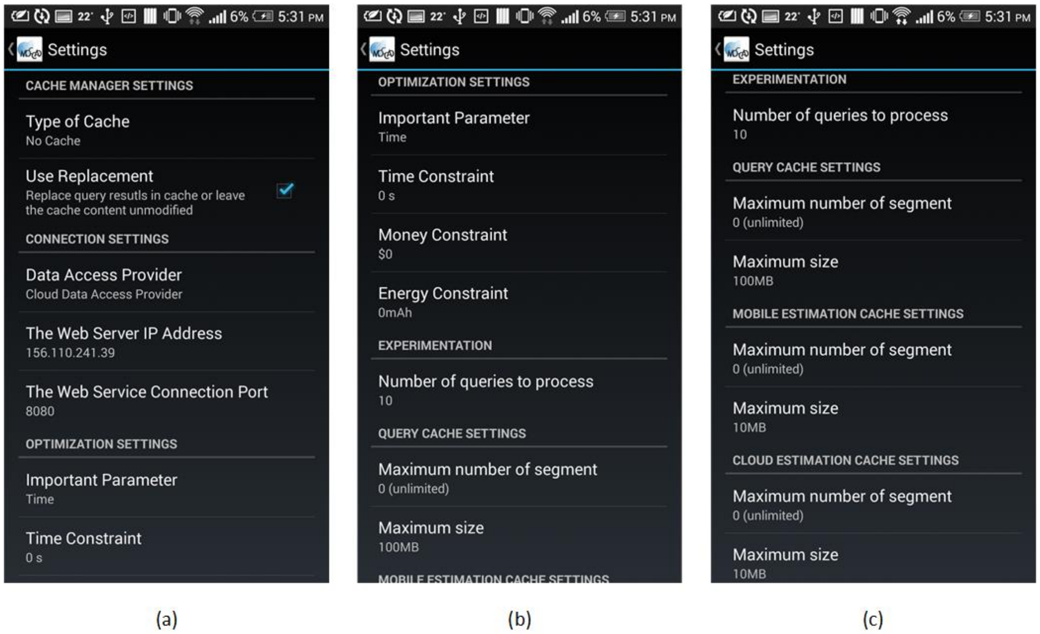
Please wait while your query result is being retrieved

```
2528 12706 7155
138307 ptogearuf uefbejzgshlf 2542 23341 30650
138450 bcsiadihyrprvj nhswkmzfjxo 2507 49072 52928
138525 dhaxkhkqqeyua wpwycmvfhbgnrnm 2549 3034 13553
138714 plfqxkzobwyxck uyemgjeqdjxrvpaoe 2511 31291 23786
139058 gyqxdewqknxknfi oqttmvwtibuof 2540 20674 97343
139115 vfevlyevkmwfs oqqjmpqrnmraykaaa 2528 38865 19652
139206 vhsbjlyygkfxybc rgikwczgbexwgdj 2546 33611 2131
139453 zifjijxraidkxk cwaceuljryhdr 2525 29622 61173
139540 klldyutljepgczyrtj bhdiyntddf 2523 34161 32796
139600 pldzvrfjfrtfdyx khsbdqndisgqps 2532 7167 15299
139704 evprpdaspnj sbjibrkqkxuty 2544 20878 29965
```

Figure 31 - Query Processing on the Prototype

The user can specify a table and add different predicates being {X op C} or {X op Y} predicates. To process the query, the user pushes the "Launch" button. The waiting message appears while the query is being processed (Figure 31.b). When the result appears, the user can see the result as a list (Figure 31.c). Each tuple can be displayed by accessing a more detailed view when clicking on the list item.

Figure 32 represents the Settings Activity. It is the interface the user can use to specify the type of cache to be used, the web service to be accessed or if the result should be replaced in the cache in the case that the cache's maximum size is reached (Figure 32.a), constraints, the parameter to minimize during optimization (processing time, energy consumption or monetary cost) or the number of queries to processed for each experimentation (Figure 32.b) and the maximum sizes of the different caches (Figure 32.c). Every time the data access manager is modified in the settings, the application automatically asks for the new metadata to the new chosen web service. The user can also view the queries he has already processed and the content of each cache.



(a)

Settings

CACHE MANAGER SETTINGS

Type of Cache
No Cache

Use Replacement ☑
Replace query resutls in cache or leave the cache content unmodified

CONNECTION SETTINGS

Data Access Provider
Cloud Data Access Provider

The Web Server IP Address
156.110.241.39

The Web Service Connection Port
8080

OPTIMIZATION SETTINGS

Important Parameter
Time

Time Constraint
0 s

(b)

Settings

OPTIMIZATION SETTINGS

Important Parameter
Time

Time Constraint
0 s

Money Constraint
$0

Energy Constraint
0mAh

EXPERIMENTATION

Number of queries to process
10

QUERY CACHE SETTINGS

Maximum number of segment
0 (unlimited)

Maximum size
100MB

MOBILE ESTIMATION CACHE SETTINGS

(c)

Settings

EXPERIMENTATION

Number of queries to process
10

QUERY CACHE SETTINGS

Maximum number of segment
0 (unlimited)

Maximum size
100MB

MOBILE ESTIMATION CACHE SETTINGS

Maximum number of segment
0 (unlimited)

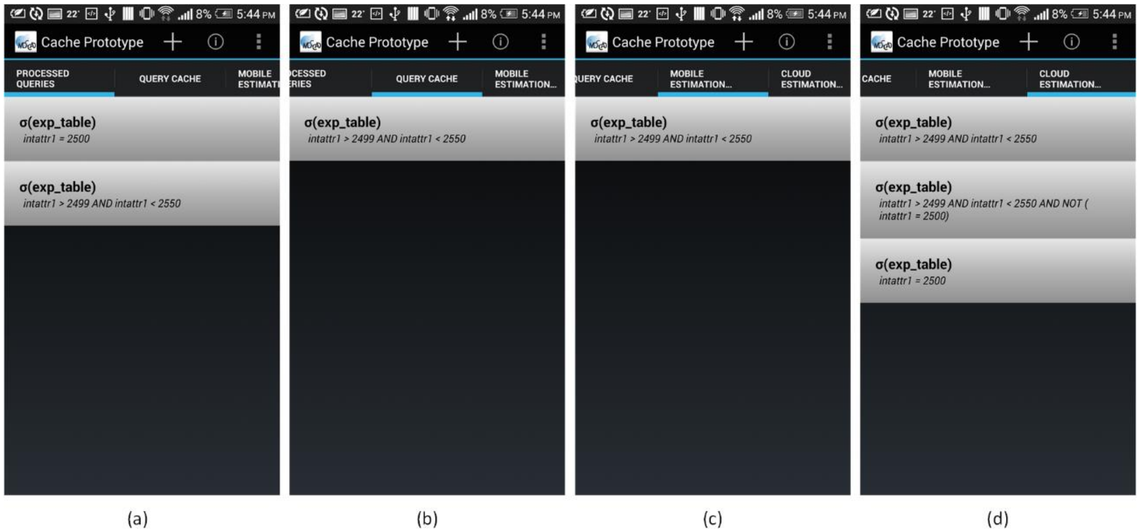
Maximum size
10MB

CLOUD ESTIMATION CACHE SETTINGS

Maximum number of segment
0 (unlimited)

Maximum size
10MB

Figure 32 - Settings on the Prototype



(a)

Cache Prototype — PROCESSED QUERIES | QUERY CACHE | MOBILE ESTIMATI…

σ(exp_table)
intattr1 = 2500

σ(exp_table)
intattr1 > 2499 AND intattr1 < 2550

(b)

Cache Prototype — …CESSED …RIES | QUERY CACHE | MOBILE ESTIMATION…

σ(exp_table)
intattr1 > 2499 AND intattr1 < 2550

(c)

Cache Prototype — …UERY CACHE | MOBILE ESTIMATION… | CLOUD ESTIMATION…

σ(exp_table)
intattr1 > 2499 AND intattr1 < 2550

(d)

Cache Prototype — CACHE | MOBILE ESTIMATION… | CLOUD ESTIMATION…

σ(exp_table)
intattr1 > 2499 AND intattr1 < 2550

σ(exp_table)
intattr1 > 2499 AND intattr1 < 2550 AND NOT (intattr1 = 2500)

σ(exp_table)
intattr1 = 2500

Figure 33 - Processed Queries and Cached Items on the Prototype

4.2 Experimentation

To be able to validate the proposed algorithm, several types of experiments have been run using the developed prototype. The experiments aim to compare the existing semantic caching algorithm as defined in [6], a system without cache and the proposed semantic caching algorithm involving decision making and the estimation cache: MOCCAD-Cache.

4.2.1 Experimentation Context

To analyze the efficiency in terms of time, energy and money for the proposed algorithm, it needs to be compared to the existing Semantic Caching algorithm in suitable conditions. The mobile device and the cloud configuration used for the different experiments are the same as the ones used with the prototype. The database stored on the cloud is generated using a relation generator developed in C#. To generate a relation, this generator takes as an input a relation schema with the type of each attribute as well as its distribution and output a file which can be easily imported with Hadoop to build the dataset in HDFS.

Figure 34 shows the cloud architecture that has been set up for experimentation purposes. It is made of a RESTful Web Service used as a middleware between the mobile device and a Hive Server used to manage query processing on the cloud. A RESTful Web Service follows the REpresentational State Transfer software architecture. This architecture gathers some principles allowing to make the web service adaptable and

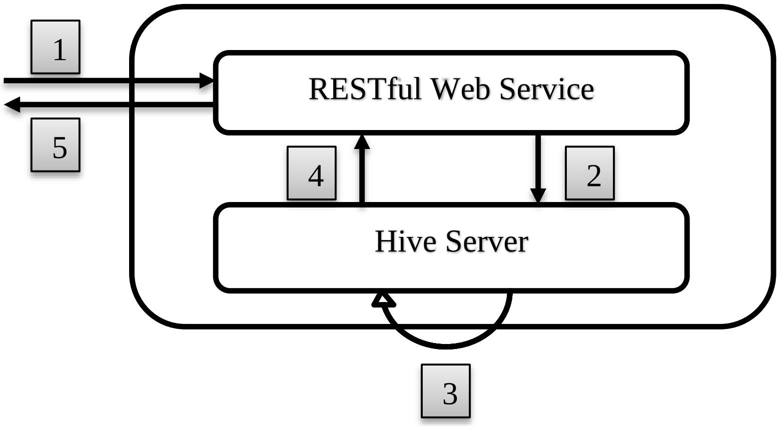


Figure 34 - Cloud Architecture for Experimentation

scalable to a high number of clients. Hive Server is a service allowing to receive HiveQL queries in order to process them on top of the Hadoop Framework. The following steps describe how this architecture works, and the different requests it can receive.

1: Request reception: The web service receives GET requests to require some information. The client (i.e. the mobile device) can request the dataset metadata, a query estimation or a query result. The query is specified in the URL as a SQL query.

2: The request is parsed and one or several consequent requests are sent to the Hive Server to build the result. If the client asked for the metadata, several requests will be sent to the Hive Server to retrieve information such as the number of tuples in the dataset, the maximum and the minimum for each attribute, etc. If the client requested an estimation, the web service will process the query several times on the Hive Server, and will store the average processing time within a collection of estimations. If the estimation is asked again for the same query, it will return the result directly. The goal of this process is to simulate a real estimation made on the cloud. Of course, it is necessary to compute all those estimations before every experiment in order to make this simulation realistic. If the client asked to process a given query, then this query is transformed in the HiveQL language and sent to the Hive Server. In the current implementation, no translation is needed from the original SQL query since HiveQL follows the SQL-92 standards. In further implementation, it could be envisioned to receive requests in other high level languages (i.e. Pig Latin).

3: The different queries sent from the web service are processed on the dataset in order to retrieve the result. The dataset schema is detailed in Table 3.

4: The retrieved result is sent back to the web service. It is then serialized under the JSON format. In the case of a query processing request, the time and money spent is also added to the query result. Since the monetary cost is also simulated, a simple cost model has been used to return the monetary cost regarding the processing time. In those experiments, it corresponds to the following formula:

$$C = C_i * n * t \tag{18}$$

With:

- C_i: the monetary cost per hour in dollars and per instance, admitting that we use only one type of instance,
- n: the number of instances (several instances can be simulated by simply dividing the processing time on one instance by the number of instances).
- t: the time in hours

5: The serialized request result is finally sent back to the client.

Table 3 - Experimentation Table Schema

Attribute Name	Type	Minimum Value	Maximum Value	Distribution
Id	big integer	1	{Number of tuples in the relation}	Sequential
Name	string			Normal (mean=15, variance=5)
Info	string			Normal (mean=15, variance=5)
Intattr1	integer	0	10000	Uniform
Intattr2	integer	0	50000	Uniform
Intattr3	integer	0	100000	Uniform

Table 3 presents the relation used for the different experiments. This relation contains mainly integer attributes in order to respect more easily the constraints brought by the satisfiability and implication algorithm [18]. The string values contain only letters from a to z in lower cases. A query generator can also create a set of queries matching different criteria such as the type of attribute used in the predicates, the maximum size of the query result and the number of queries to be generated. Once those queries are generated, they are analyzed and sorted out as exact hits, extended hits or partial hits for a given set chosen as the query cache content. Finally, the queries are gathered into different query sets and used to analyze the performance of our algorithm.

4.2.2 Performance Metrics

The goal of the different experiments is to measure the performance of the MOCCAD-Cache algorithm based on different metrics:

Total Processing Time:

When testing a cache algorithm used within a query processing system, the goal is to show its efficiency in terms of time. Indeed, in our algorithm, since some decisions have to be made before processing a query, and those decisions can have some overhead, it is important to verify that these decisions impact the total time to process a query. Of course, depending on the type of cache hit the decisions are not the same. Therefore, the overhead can be different and the difference in term of improvement relatively to other algorithms needs to be analyzed.

Total Monetary Cost:

Time, however, cannot be studied by itself. Indeed, one can think that since processing a query on the mobile device does not cost any money, it is always better to process it on the cache. That is true if we consider only money. Nevertheless, we could indeed imagine that one just pays for a very powerful cloud infrastructure, and would succeed in processing queries way faster on the cloud than on the mobile device (assuming that the result is available at both places). Thus, the MOCCAD-Cache would be efficient in terms of time since it would allow the user to take advantage of its efficient cloud. However, it is not efficient in terms of money cost since a lot of money would have been spent for an improvement in time efficiency that might not be much comparing with a standard semantic caching algorithm. Consequently, it is more about observing the total monetary cost that would bring some time efficiency compared to other cache algorithms than observing money or time by itself.

Total Energy Cost:

The goal of our algorithm is also to be aware of energy. When the algorithm finds that the cloud might be more efficient to run a query in terms of time, it is likely the algorithm might cost more energy due to the energy consumed by the mobile device when it uses network compared to when it processes operations locally. However in some cases where, for example, the computation on the mobile device is tremendous, retrieving only a few data items from the cloud can be cheaper in terms of energy. Here again, the query would also be more expensive and can also be slower to process.

In order to control every experiment to be able to analyze the result, Table 4 and Table 5 list the different parameters used in the experiment. The static parameters table contains all the parameters which will not be modified throughout each experiment. It specifies the currents for each state of each component. Those parameters can be found on an Android device via a Power Profile. The average bandwidth needs to be measured right before an experiment so that the estimations related to processing queries on the cloud get as realistic as possible. Concerning the *exp_table* table, the maximum tuple size is retrieved from the web service each time the data access provider is modified in the settings. This tuple size is used to estimate the total size of a query result and thus the cost to retrieve it from the cloud to the mobile device. In order to control this experiment, the number of attributes in each predicate and the number of inequalities are defined for every query in each query set used for the experiment. For example, using a random number for the number of inequalities can cause some experiment query set to be much different than others in terms of cache analysis. Indeed, let us assume that one experiment's query set has an average of 10 inequalities per query whereas another one has an average of 5 inequalities per query. In this case it can be difficult to analyze any result since in the first case, the time spent for the cache analysis can be more important. Even though the cache analysis time relative to the whole processing time is small, some other parameters such as the size of the result (Query Size, Table 5) can bring some variation as well. Therefore, making this parameter static makes the analysis of the result easier.

Table 5 specifies the different parameters that will be varied in different experiments in order to study their impacts on the performance of the MOCCAD-Cache algorithm.

Table 4 - Static Parameters

Static Parameter	Value(s)	Reference
SoC CPU Active Mode Freq	1.728 GHz	Kernel
SoC CPU Active Mode Current	162.00 mA	Power profile
SoC CPU Idle Mode Freq	0	Kernel
SoC CPU Idle Mode Current	2.52 mA	Power profile
SoC Wi-Fi Network Low Current	3.6 mA	Power profile
SoC Wi-Fi Network High Current	74 mA	Power profile
Battery Capacity	2300 mA	Power profile, HTC
Number of Instances Simulated on the Cloud	5	
Average Bandwidth (to be measure right before experimentation)	15Mbps/14Mbps	Speedtest Android Application…
Maximum Tuple Size	49 Bytes	Computation on exp_table table
Number of different variables in each predicate	1	
Number of inequalities in each predicate	2	
Maximum Number of Cache Entries in Query Cache	10	
Maximum Number of Cache Entries in Estimation Cache	+Inf	
Query Cache Maximum Size	100 MB	
Mobile Estimation Cache Maximum Size	10 MB	
Cloud Estimation Cache Maximum Size	10 MB	
Query Set size	50 queries	
Number of Relations	1	
Relation Size	200,000	

Table 5 - Dynamic Parameters

Parameter	Value Range	Default Value	Reference
Cache types	{No Cache, Semantic Cache, MOCCAD-Cache}	MOCCAD-Cache	
Query Size (% of the relation)	[0%;75%]	N/A	Phone constraint on the maximum heap size available per application (200 MB)
Query Cache Exact Hit Percentage (%)	[0%;100%] for each 10%	0%	
Query Cache Extended Hit Percentage (%)	[0%;100%] for each 10%	0%	
Query Cache Partial Hit Percentage (%)	[0%;100%] for each 10%	0%	

In order to study the performance of the MOCCAD-Cache in each case where we can use some result from the cache, the different experiments aim to measure the total processing time, the total monetary cost and the total energy consumption for each percentage of exact hit, extended hit and partial hit 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.

Table 6 presents the size of each query contained in the cache for all the presented experiments. The query results contained in cache do not overlap.

Table 6 - Cached Queries' Result Size

Cached Queries	Approximated Result Size (Number of Tuples)	Percentage of the relation
SELECT * FROM exp_table WHERE id >= 0 AND id <= 94256;	94257	47.12 %
SELECT * FROM exp_table WHERE id > 166258 AND id <= 199523;	33265	16.63 %
SELECT * FROM exp_table WHERE intattr1 > 9554 AND intattr1 <= 10000;	9820	4.91 %
SELECT * FROM exp_table WHERE intattr1 >= 1234 AND intattr1 <= 2326;	21860	10.93 %
SELECT * FROM exp_table WHERE intattr2 >= 3250 AND intattr2 <= 4465;	4864	2.43 %
SELECT * FROM exp_table WHERE intattr2 >= 42640 AND intattr2 < 48256;	22464	11.32 %
SELECT * FROM exp_table WHERE intattr3 > 75414 AND intattr3 < 80000;	9170	4.58 %
SELECT * FROM exp_table WHERE intattr3 > 42 AND intattr3 < 15564;	31044	15.52 %
SELECT * FROM exp_table WHERE intattr1 >= 1000 AND intattr1 < 1233;	4660	2.33 %
SELECT * FROM exp_table WHERE intattr2 >= 63 AND intattr2 <= 100;	152	0.08 %

4.2.3 Results

This subsection shows the results acquired from the experiments ran with the different metrics discussed in the previous section. It validates the proposed solution by showing that MOCCAD-Cache improves the processing time but however is more expensive in terms of monetary cost when the objective is to minimize time. Due to experimental issues and consequently a lack of time, no experiments have been done to show that with MOCCAD-Cache, the user can also choose to minimize monetary cost. Some points are however discussed towards this aspect.

4.2.3.1 Impact of the Exact Hit Percentage on the Query Cost

In order to study the performance of the MOCCAD-Cache regarding the exact hit percentage on the query cache, it is necessary to fix the database relation to be used. As specified in Table 5, the exp_table relation stores 200,000 tuples. Then 10 queries are chosen to warm up the query cache before this experimentation. No replacement is being done during the experimentation. 10 query sets are generated containing fifty queries to be processed so that they match each percentage of cache exact hit and cache miss. Each query set owns a percentage of queries matching an exact hit and a remaining percentage of queries matching a cache miss. For example when the set of queries needs to match 50% of exact hit on the query cache, there are 25 queries matching an exact hit and 25 queries matching a cache miss. The estimation caches are also warmed up before the experiments.

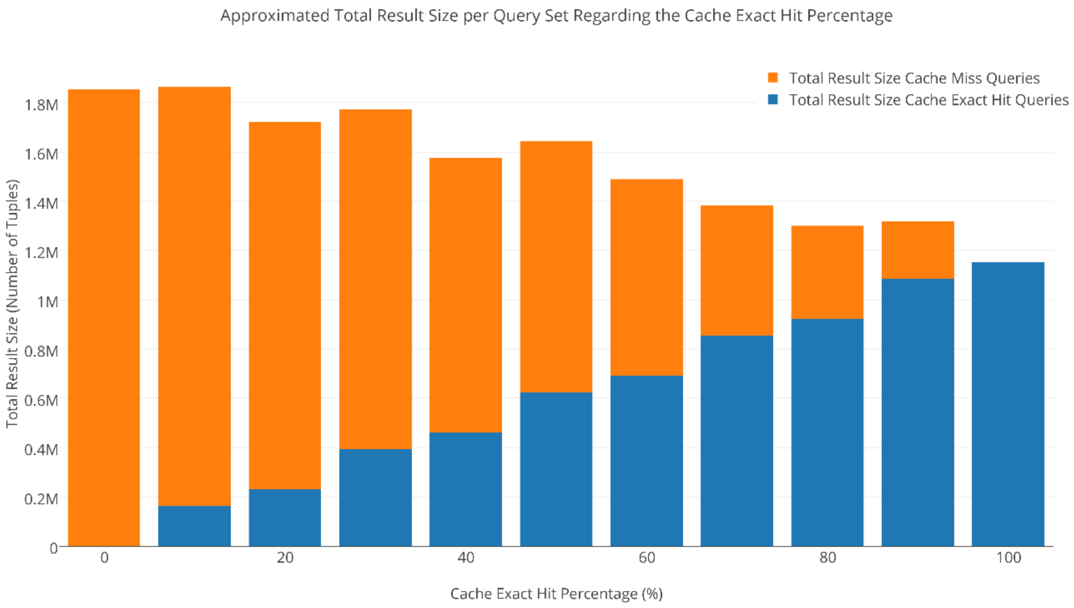


Figure 35 - Approximated Total Result Size per Query Set Regarding the Cache Exact Hit Percentage

Figure 35 shows the total result size for each query set. Indeed, by computing the size of each query (the same way it has been explained in section 2.2.1), the total result size per query set can be shown. For example, when 0% of the query set match an exact hit, processing all the query set corresponds to the retrieval of approximately 1.82 Million tuples. In blue, the result size corresponding to the queries matching an exact hit increases when the percentage of exact hit goes up. In Orange, the result size corresponding to the queries matching a cache miss decreases when the percentage of cache miss goes up. The total result size per experiment, however, globally decreases when the percentage of exact hit increases. This chart can be used to analyze the results of the query costs and potentially be able to deduce some trending on this cost regarding the cache hit percentage. This decrease in size is due to the fact that the queries matching an exact hit and a cache miss do not have the same average result size. This is related to the queries chosen as cached queries (Table 6). For example, an exact hit cannot match more than 47.12 % of the relation. However, a cache miss can match a maximum of 99.92 % of the relation.

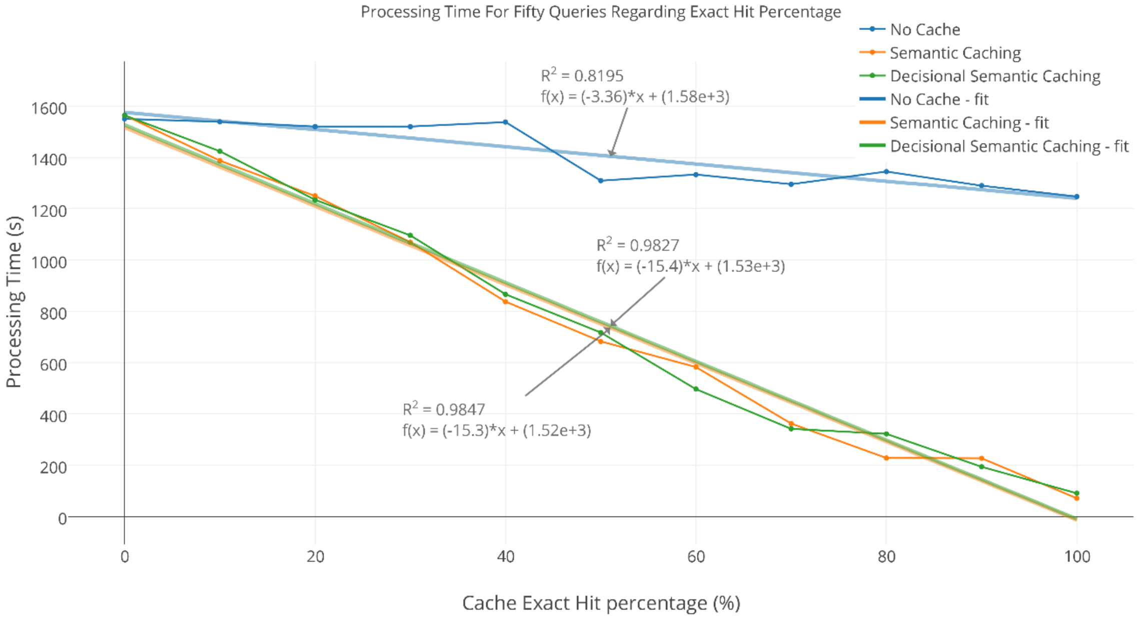


Figure 36 - Processing Time for Fifty Queries Regarding Cache Exact Hit Percentage

Figure 36 shows the average processing time for each experimentation. On this chart we can observe that the two curves corresponding to the semantic caching algorithm and the MOCCAD-Cache algorithm overlap. For those two curves, the processing time decreases linearly when the percentage of exact hit increases. The third curve, describing the processing time in the case where there is no cache, decreases a bit and can be explained by the global diminution in the total result size presented in Figure 35.

When using semantic cache or a MOCCAD-Cache, the processing time regarding the percentage of exact hit is similar since in both cases no estimation is being made and the query is being processed directly on the mobile device when this query matches an exact hit. Thus, semantic cache and MOCCAD-Cache are as efficient in the case of an exact hit in terms of processing time.

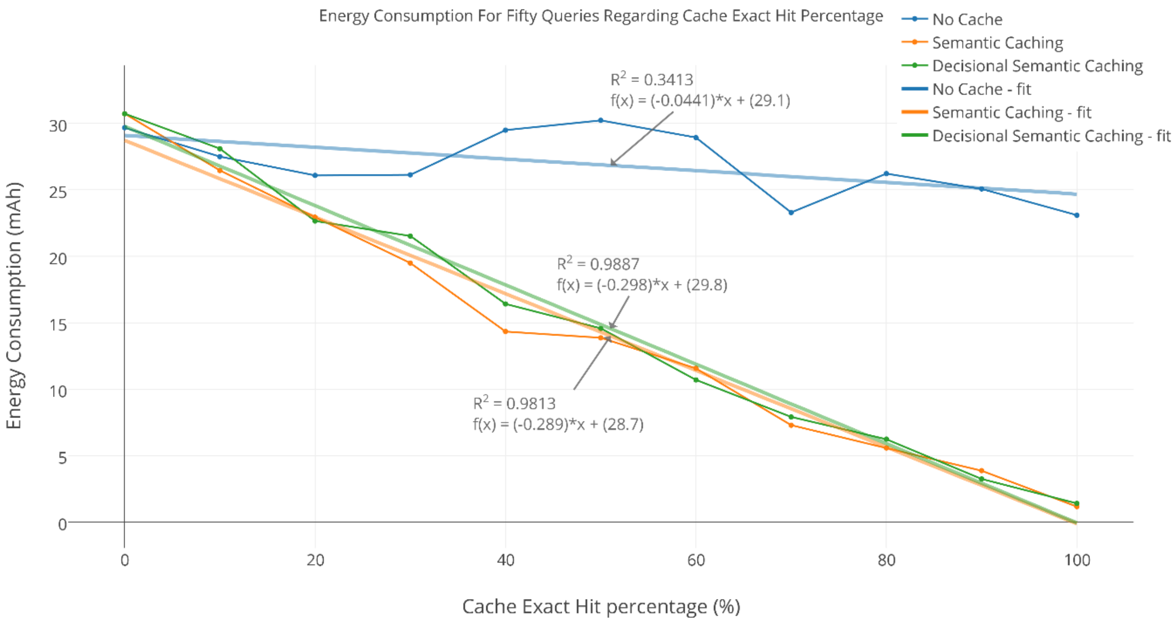


Figure 37 - Energy Consumption for Fifty Queries Regarding Cache Exact Hit Percentage

Figure 37 presents the variations of energy consumption regarding the cache exact hit percentage and follows more or less the variations of processing time regarding the cache exact hit percentage and can also be explained by the similarity of the semantic caching and the MOCCAD-Cache algorithms in this situation. Also, the energy spent in the case where no cache is being used decreases and depends as well on the processing time. Therefore, semantic caching is similar to MOCCAD-Cache concerning energy consumption regarding the percentage of exact hit.

Figure 38 shows the money spent for fifty queries regarding the query cache exact hit percentage. Here again, the semantic cache and the MOCCAD-Cache are similar in terms of money cost and decrease when the percentage of exact hit increases since less queries are processed on the cloud in both cases.

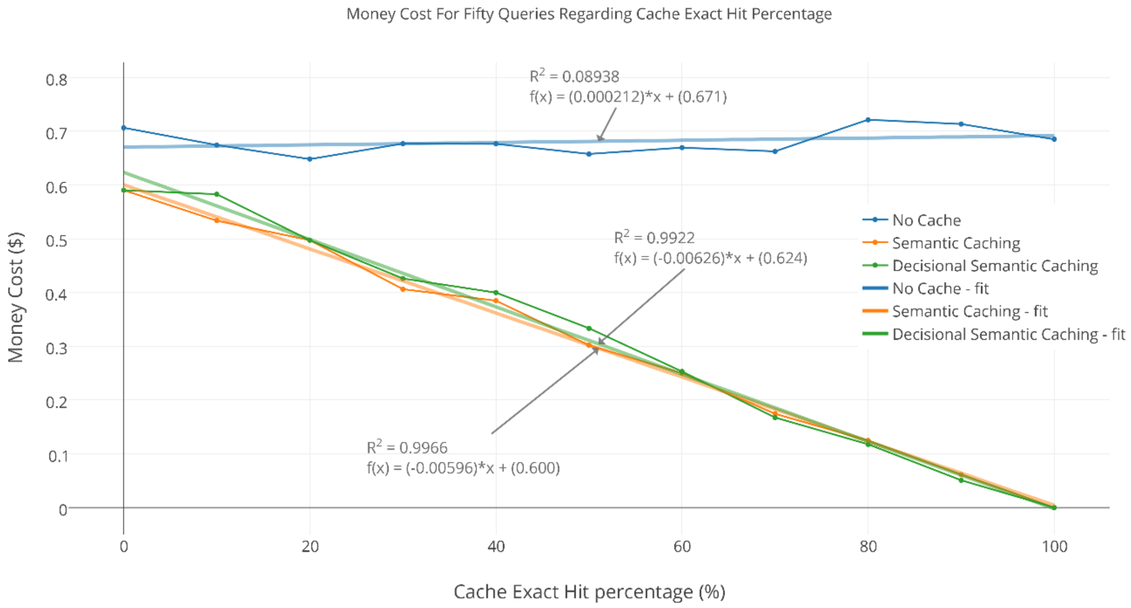


Figure 38 - Money Cost for Fifty Queries Regarding Cache Exact Hit Percentage

When there is no cache the money spent remains approximately the same all along the experiments. This is explained by the fact that the total processing time on the cloud remains also the same for the experimentation (Figure 39) due to the used cost model exposed in Section 4.2.1.

Finally, we can see that the MOCCAD-Cache algorithm is as efficient as the semantic cache algorithm in the case of a query cache exact hit. This is allowed by the estimation cache which prevents the overhead of computing an estimation for the experiments' cache miss queries.

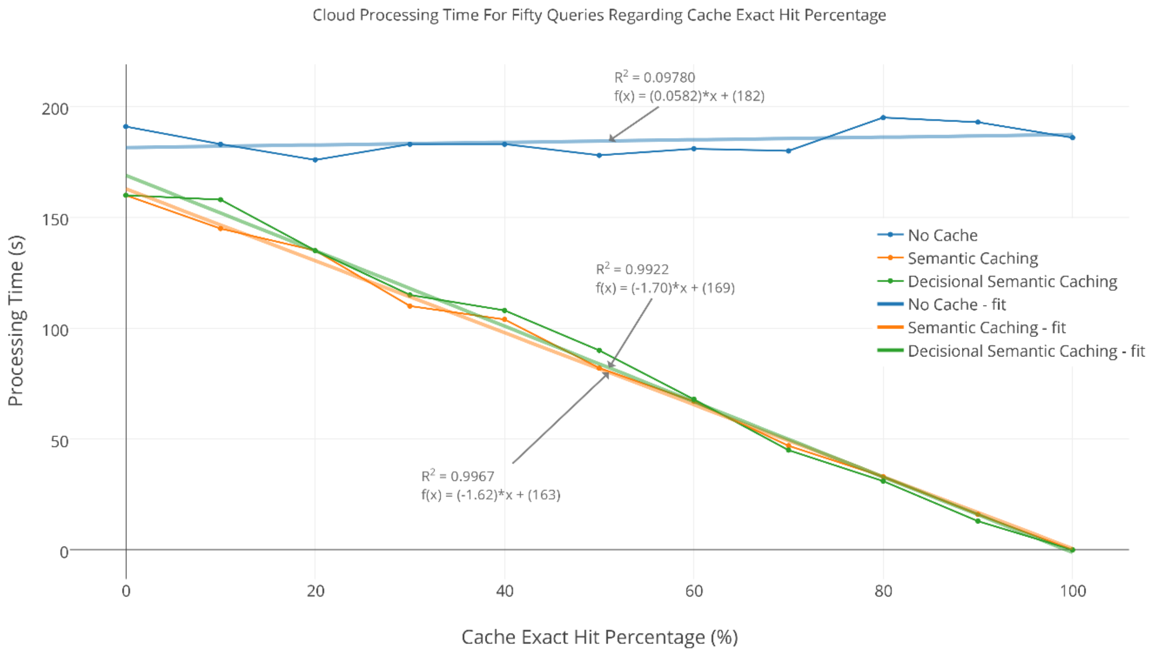


Figure 39 - Cloud Processing Time for Fifty Queries Regarding Cache Exact Hit Percentage

4.2.3.2 Impact of the Extended Hit Percentage on the Query Cost

With the same approach than exact hit, a 200000-tuple database is being used. Also, the same ten queries are used to warm up the cache. No replacement in the query cache is done during the experimentation. The query sets are also built so that they match the percentage of cache extended hit as well as the remaining percentage of cache miss against the given query cache. The estimation caches are also warmed up by preprocessing those queries.

Figure 40 shows the total result size for each query set. For the given query sets, the size of the query sets corresponding to the extended hit queries, in blue, increases linearly. In orange the size of the query sets corresponding to the cache miss queries decreases linearly. Also, the total query size globally decrease when the extended hit percentage increases.

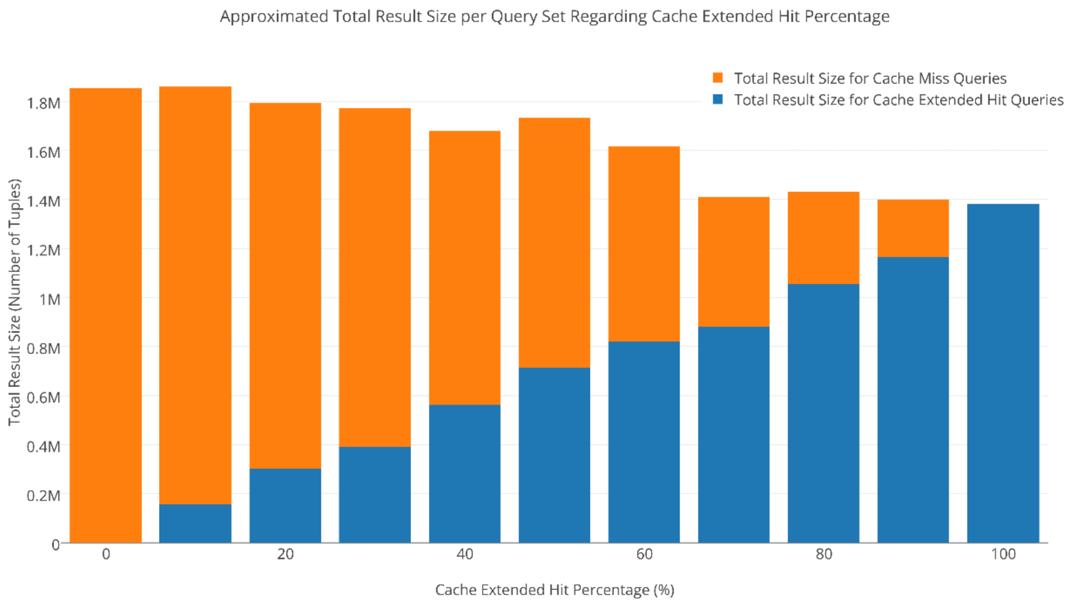


Figure 40 - Approximated Total Result Size per Query Set Regarding Cache Extended Hit Percentage

Figure 41 shows the processing time regarding the query cache extended hit percentage when there is no cache (blue), a semantic cache (orange) or our MOCCAD-Cache (green). When no cache is being used, the processing time decreases following the total result size of the used query sets (Figure 40). The processing time, when using a semantic cache, decreases faster than when no cache is being used. Indeed the processing time follows the decrease in the total result size for the query cache misses but also uses some processing time for the extended hit queries processed on the mobile device. Therefore, it takes additional time to process the queries when no cache is used compared to the semantic cache and the MOCCAD-Cache.

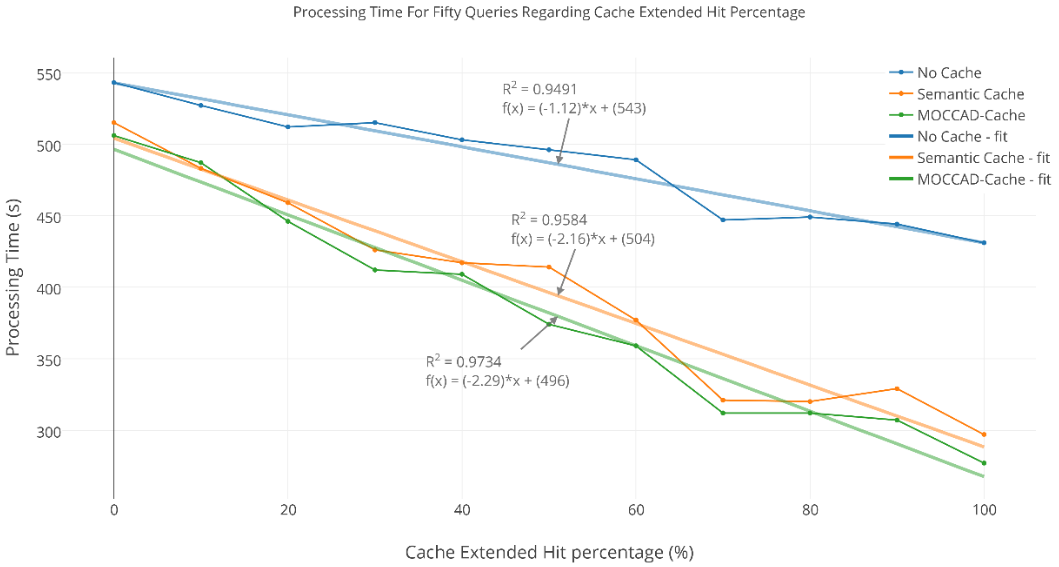


Figure 41 - Processing Time for Fifty Queries Regarding Cache Extended Hit Percentage

When using our MOCCAD-Cache, the processing time also logically decreases regarding the percentage of extended hit. However, we can see that the processing time, when we use MOCCAD-Cache, is globally reduced compared to the processing time when using a semantic cache. This can be explained by looking at Figure 42 which shows the percentages of queries that are processed on the cloud regarding the percentage of extended hit. Indeed, we can see that with a semantic cache, all the queries matching an extended hit are processed on the mobile device. However, we can see that with MOCCAD-Cache, more queries are processed on the cloud than with a semantic cache (Figure 42). This shows that the MOCCAD-Cache's optimizer decided to reprocess some queries on the cloud rather than processing them on the mobile device after it computed the different estimations. This can thus explain the globally lower processing time when using the MOCCAD-Cache than when using semantic caching.

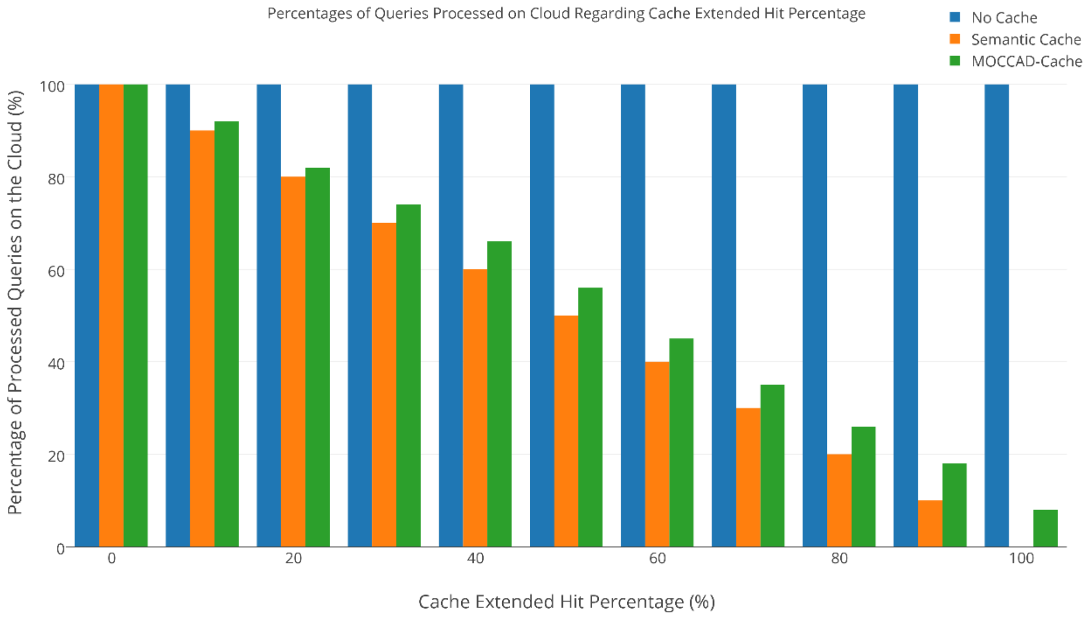


Figure 42 - Percentages of Queries Processed on Cloud Regarding Cache Extended Hit Percentage

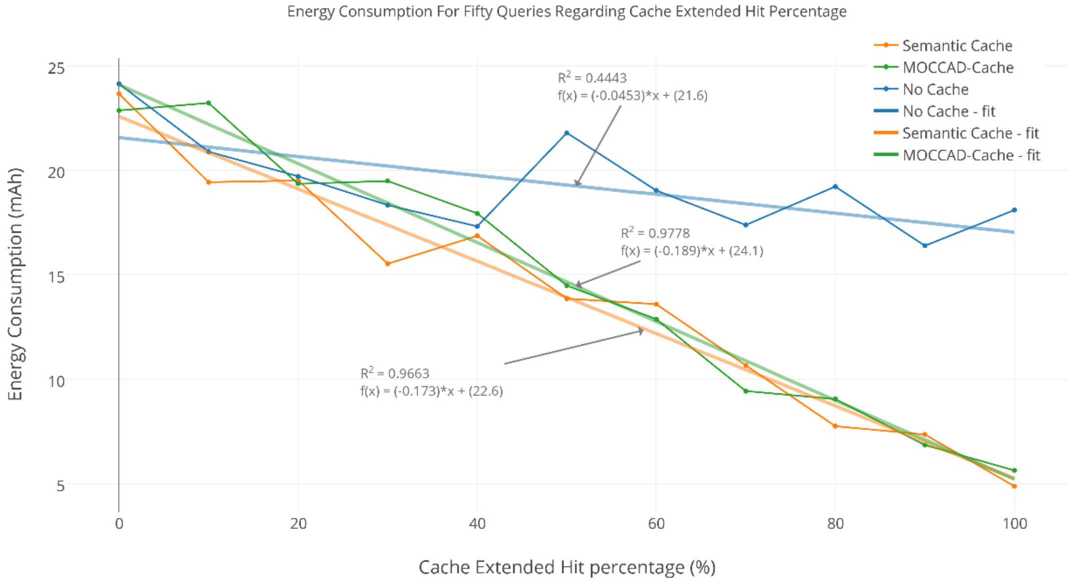


Figure 43 - Energy Consumption for Fifty Queries Regarding Cache Extended Hit Percentage

Figure 43 shows the variation of energy consumption regarding the cache extended hit percentage. When using a semantic cache, the energy decreases regarding the percentage of extended hit. This is explained by the fact that less queries are processed on the cloud when the extended hit percentage increases. Indeed, using the network interfaces is expensive in terms of energy consumption since it can take some time to download the result, and that energy depends on that time (Formula 13). In addition, even though more queries are processed on the cloud with MOCCAD-Cache when the extended hit percentage increases (Figure 42), the energy spent when using MOCCAD-Cache tends to be similar to the energy spent when using a semantic cache. This is explained by the fact that the queries that are chosen to be reprocessed on the cloud do not retrieve a lot of data (Figure 44). Thus, by looking at the current used for each state in Table 4, we can see that the current to download data from the cloud (SoC CPU Active Mode Current) is lower than the current to process operations on the mobile device (SoC Wi-Fi Network High Current). This shows that if query result size is low, the time and the energy consumption to process and download data from the cloud can be lower than processing a huge segment, as discussed in Chapter 2. Therefore, Figure 43 and Figure 44 show that the energy consumption is saved by processing queries with small results on the cloud. When no cache is being used, all the queries are processed on the cloud. Therefore, the curve corresponding to this situation is above the others in Figure 43 meaning that not using a cache consumes more energy when the percentage of extended hit increases.

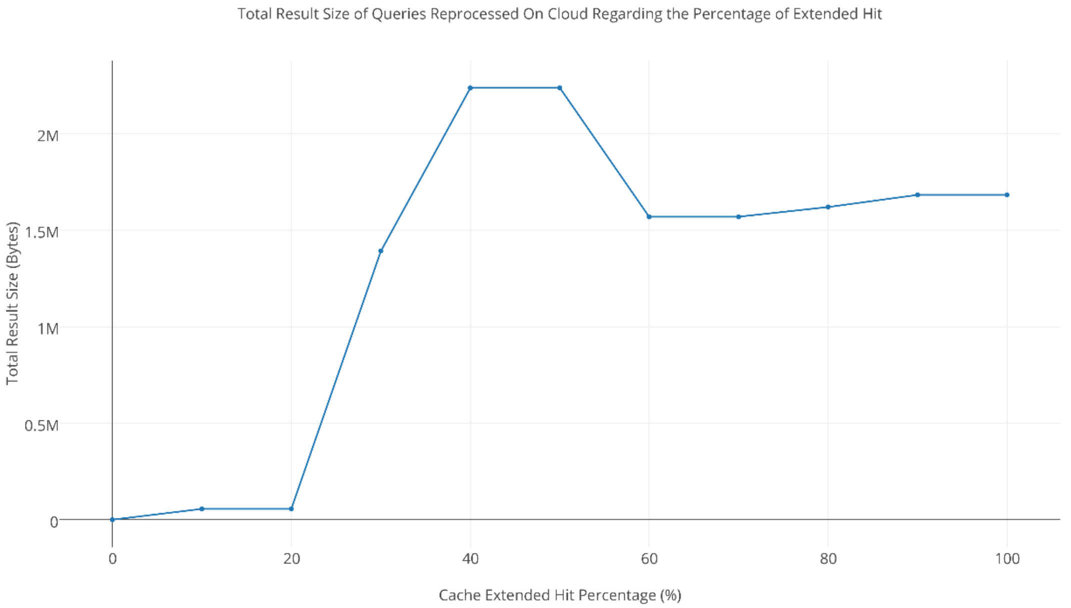


Figure 44 - Total Result Size of Queries Reprocessed on Cloud Regarding the Percentage of Extended Hit

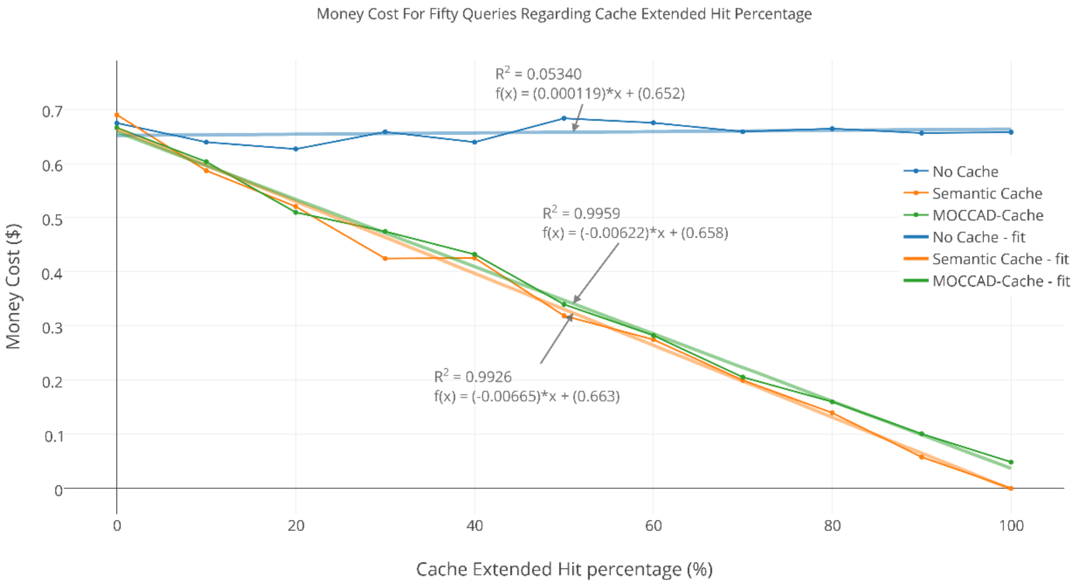


Figure 45 - Money Cost for Fifty Queries Regarding Cache Extended Hit Percentage

Figure 45 shows the money spent for fifty queries regarding the percentage of extended hit. When no cache is being used, all the queries are processed on the cloud (Figure 42) and thus becomes expensive in terms of monetary cost. When a semantic cache or MOCCAD-Cache is being used, the amount of energy spent on the cloud is reduced. However, MOCCAD-Cache tends to consume a little bit more money than a semantic cache when the percentage of extended hit increases. Indeed, since more queries are processed on the cloud when using the MOCCAD-Cache (Figure 42), the amount of money spent is also increased when compared to the monetary costs while we use a semantic cache.

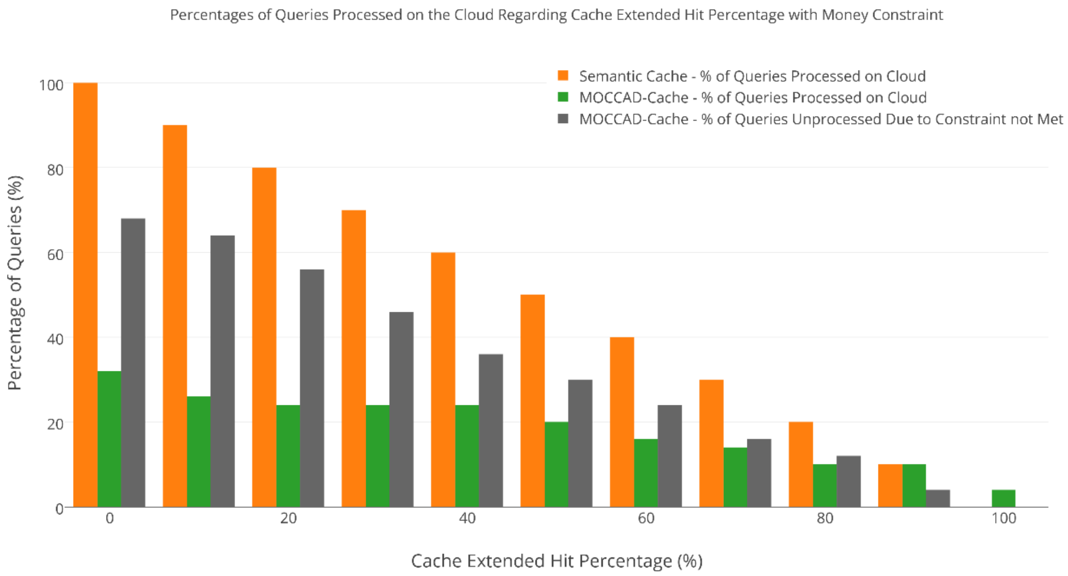


Figure 46 - Percentages of Queries Processed on the Cloud Regarding Cache Extended Hit Percentage with Money Constraint

In addition of minimizing the processing time with the considered parameters, we have added a money constraint of $0.015 per query. When MOCCAD-Cache estimated the possible costs to process the query, and that it occurred that none of the possibility met the user’s money constraint, then the query is not processed. Figure 46 shows the percentages of queries that have been processed on the cloud with Semantic Caching and with MOCCAD-Cache as well as the percentage of queries that have been chosen not to be processed at all with the MOCCAD-Cache. We can see that with the semantic caching algorithm, all the queries matching a cache miss are processed on the cloud and the queries matching an extended hit are processed on the mobile device. This result is exactly the same as the one showed in Figure 42. However with MOCCAD-Cache, some queries matching an extended hit, are processed on the cloud (100% extended hit) and some queries are not processed because they have been foreseen not to respect the user's constraints.

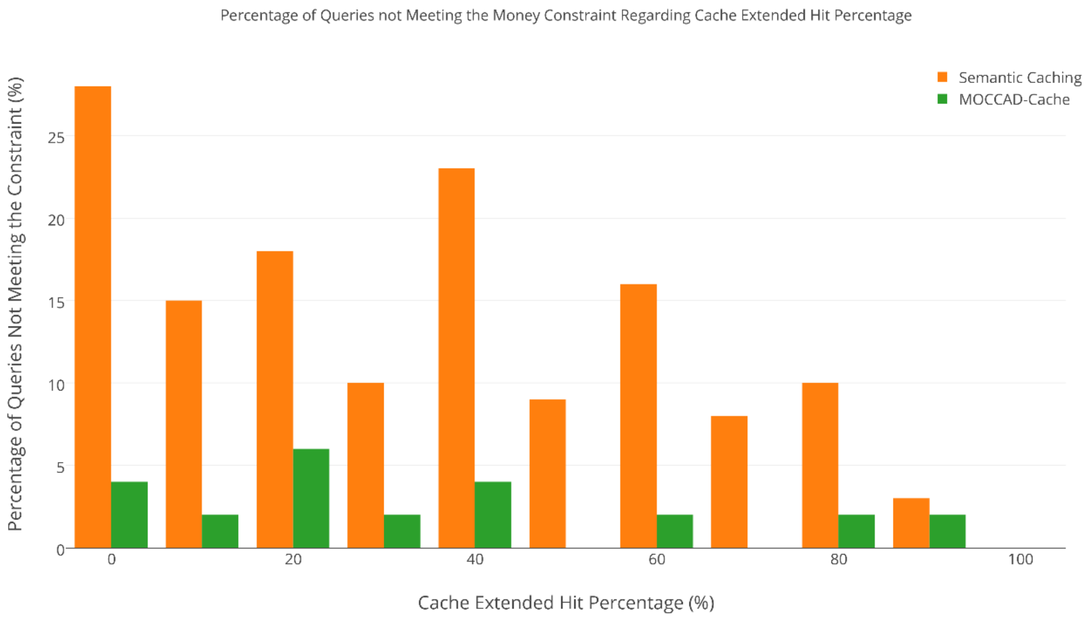


Figure 47 - Percentages of Queries not Meeting the Money Constraint Regarding Cache Extended Hit Percentage

After processing the queries, Figure 47 demonstrate that MOCCAD-Cache can take into account the user constraints whereas a standard semantic caching algorithm is unable to do the same. Indeed, Figure 47 shows that much more queries do not respect the user's constraints with semantic caching whereas only a few of them do not respect the user's constraints with MOCCAD-Cache. Those few errors are due to the inaccuracy of the computed estimations. Therefore some queries may have been estimated as meeting the constraints before query processing but actually did not meet the constraints once processed with the chosen query plan.

Finally, the MOCCAD-Cache algorithm is more efficient than the semantic caching algorithm in terms of processing time, remains equivalent in terms of energy, but however consumes a little bit more money due to the number of queries processed on the cloud. The close results between the semantic cache and the MOCCAD-Cache can first be explained by the fact that a small cloud with only 5 instances has been used. If the cloud gets faster, more queries will be processed on the cloud rather than on the mobile device. Therefore, the difference between MOCCAD-Cache and the semantic cache would be more significant. Secondly, the throughput variation between the mobile device and the cloud can lead the mobile device to process the query on the cloud with the computed estimation. However, during the execution, the query may take longer to be processed on the cloud than it would have taken to be processed on the mobile device. If the user specifies a constraint such as money, our algorithm is able to estimate whether a given query will meet this constraint and thus decide or not to process it. Considering the other parameters, if the parameter to be optimized is money, the query will be processed on the mobile device since it is free of charge.

4.2.3.3 Impact of the Partial Hit Percentage on the Query Cost

Figure 48 shows the total result size of the query sets used for those experiments regarding the percentage of partial hit. The result size corresponding to the cache partial hit queries, in blue, increases when the percentage of partial hit increases. The result size corresponding to the cache miss queries, in orange, decreases when the partial hit percentage increases. The total result size per query set increases when the partial hit percentage increases.

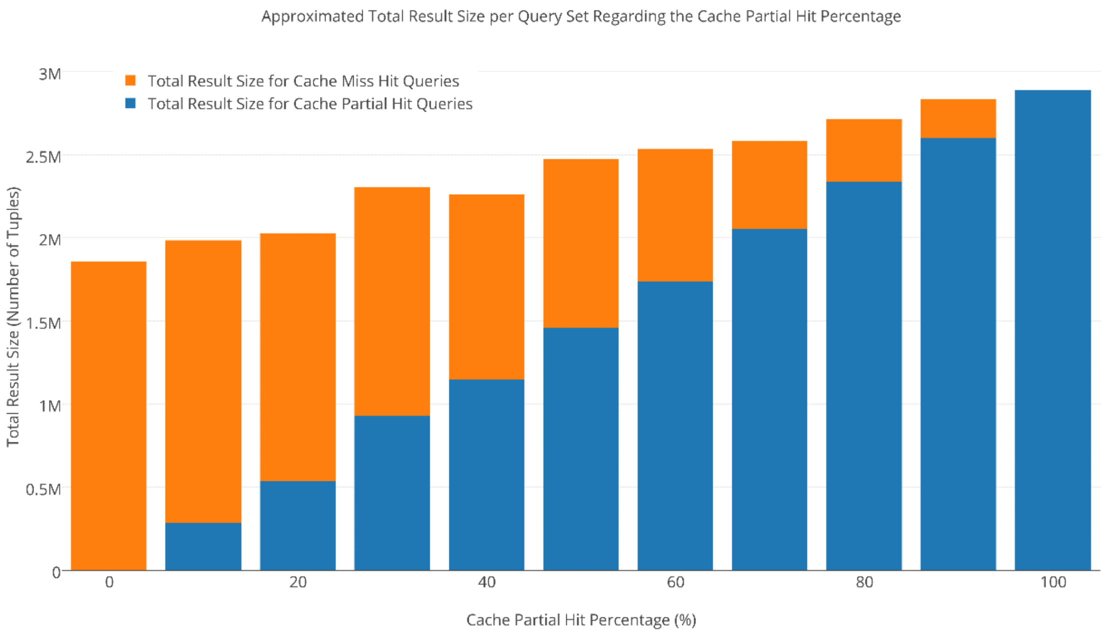


Figure 48 - Approximated Total Result Size per Query Set Regarding the Cache Partial Hit Percentage

Figure 49 shows the percentage of queries that have been processed on the mobile device regarding the cache partial hit percentage. We do not show any results related to the percentage of queries that have been processed on the cloud since in the case of a partial hit, there is always a query processed on the cloud. However, it can either be the input query or a remainder query. From this figure, we can see that as much queries are processed on the mobile device with a semantic caching algorithm as with MOCCAD-Cache. This means that the decision process decided that it was always more beneficial to process the probe query on the mobile device and the remainder query on the cloud rather than processing the whole input query on the cloud.

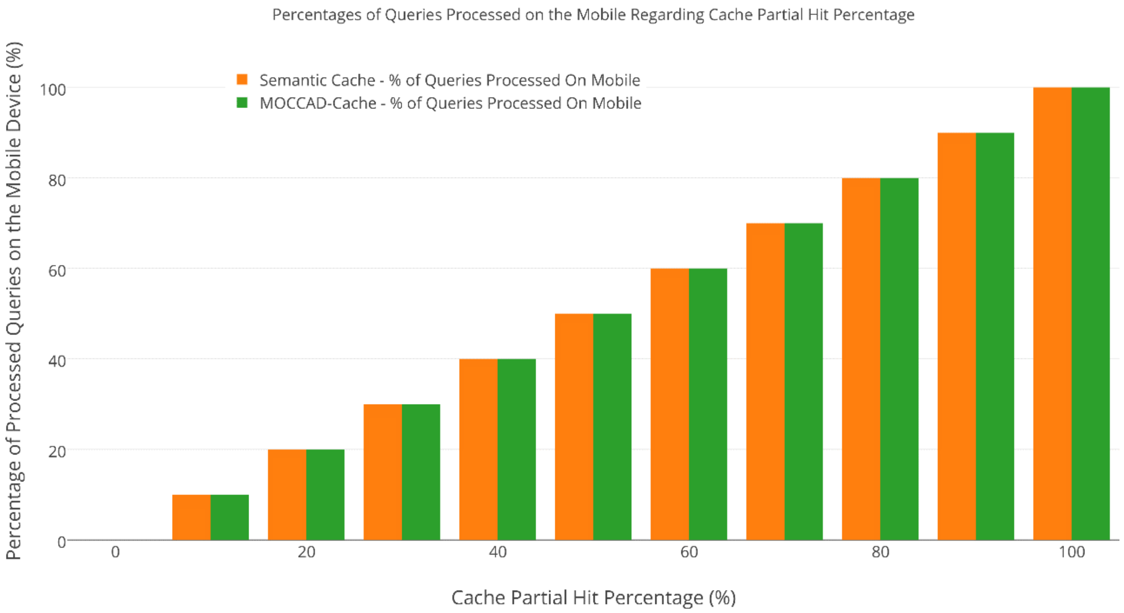


Figure 49 - Percentages of Queries Processed on the Mobile Regarding Cache Partial Hit Percentage

Figure 50 presents the processing time for fifty queries regarding the percentage of cache partial hit queries. When no cache is being used, all the queries are processed on the cloud. This explains the growth in processing time regarding the cache partial hit percentage. Concerning semantic caching and MOCCAD-Cache, when the cache partial hit percentage increases, the processing time increases as well. Indeed this is explained by the fact that the result size of the cache miss queries are globally bigger than the result size of the partial hit queries. Even though the MOCCAD-Cache algorithm seems to be more efficient than the semantic caching algorithm on this graph. However, by looking at Figure 51, we can see that the downloading time is lower with MOCCAD-Cache than with the semantic cache, even though it downloads the same amount of data in both cases.

Therefore, MOCCAD-Cache and semantic caching should be more considered as equivalent in terms of processing time.

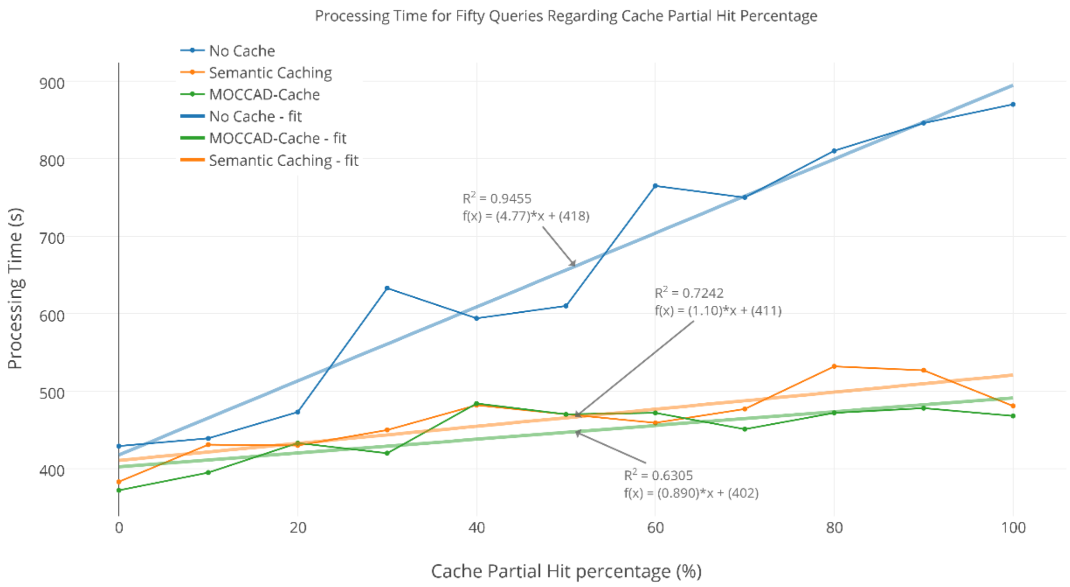


Figure 50 - Processing Time for Fifty Queries Regarding Cache Partial Hit Percentage

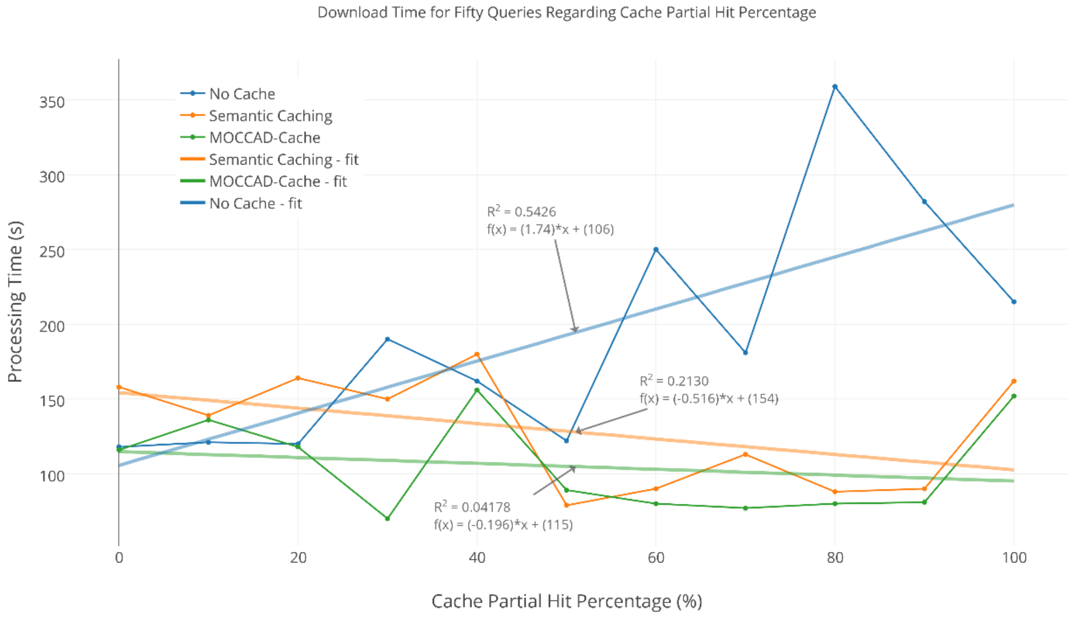


Figure 51 - Download Time for Fifty Queries Regarding Cache Partial Hit Percentage

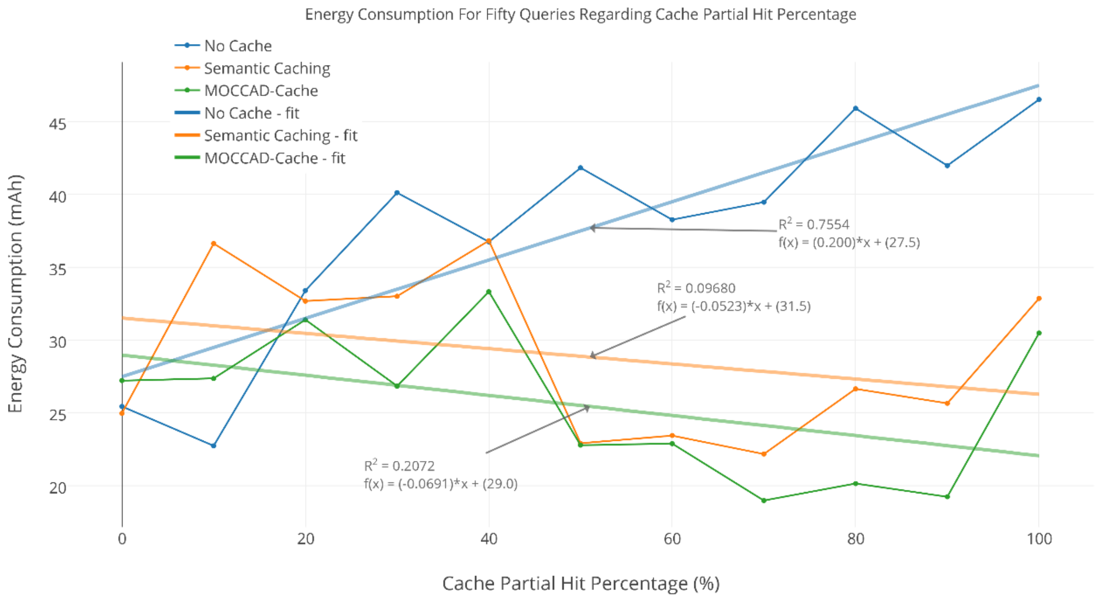


Figure 52 - Energy Consumption for Fifty Queries Regarding Cache Partial Hit Percentage

Figure 52 presents the energy consumption for fifty queries regarding the percentage of extended hit. Concerning MOCCAD-Cache and semantic caching, because the result is being downloaded faster from the cloud (Figure 51) regarding the percentage of partial hit, the consumed energy is thus reduced (Formula 13). When no cache is being used, more data is being downloaded from the cloud than when using MOCCAD-Cache or a semantic cache. Thus more time is being taken to retrieve the result. This explains that the energy consumption curve in the case no cache is being used is above the other curves. For the same reasons, since more and more data is being downloaded when the percentage of partial hit increases (Figure 48), the energy consumed is becoming more important as the percentage of partial hit gets bigger.

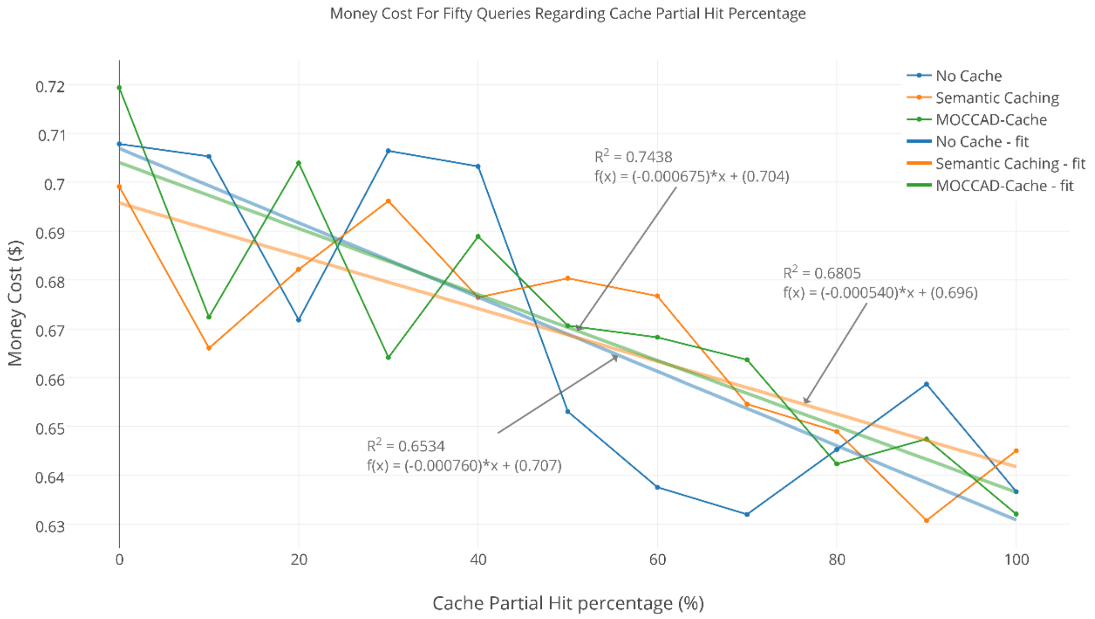


Figure 53 - Money Cost for Fifty Queries Regarding Cache Partial Hit Percentage

Figure 53 presents the monetary cost for fifty queries regarding partial hit percentage. We can see that the money spent when no cache is being used is similar to the money spent when MOCCAD-Cache or a semantic cache is being used. In the three cases, for each query posed, one query is processed on the cloud. It is either an input query or a remainder query. Therefore, the difference between the query processor without cache and the one with a MOCCAD-Cache or a semantic cache is less remarkable than the difference observed with exact hit queries.

Finally, the MOCCAD-Cache algorithm does not succeed in being better than the semantic caching algorithm. Indeed, when the cloud can perform fast queries with a high number of instances, the estimated time and energy is mainly related to the download time and the size of the result. Thus when a decision needs to be made, the algorithm chooses to process the query retrieving the least data. This corresponds to the query plan which process the probe query on the mobile device and the remainder query on the cloud.

This query plan is identical to the one used in the semantic caching algorithm and thus, no improvement is being made.

4.2.3.4 Global Impact of Cache Hit Percentage on the Query Cost

In this subsection, we validate our algorithm by showing the impact of MOCCAD-Cache on the processing time and the monetary cost and we compare it with the semantic cache. Those results corresponds to the sum of the previously presented results. Thus each value 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.

Figure 54 shows that the processing time is globally reduced when we use MOCCAD-Cache. Figure 55 shows that the user will however need to pay the price of this improvement in terms of time. Once again, the small difference 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 emphasis this difference.

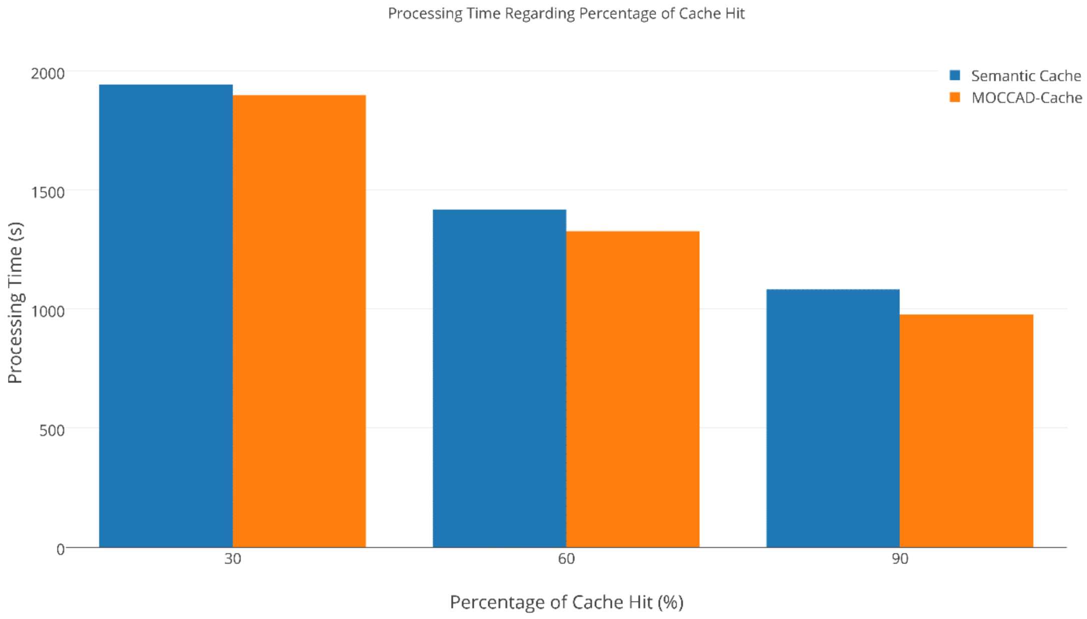


Figure 54 - Processing Time Regarding Cache Hit Percentage

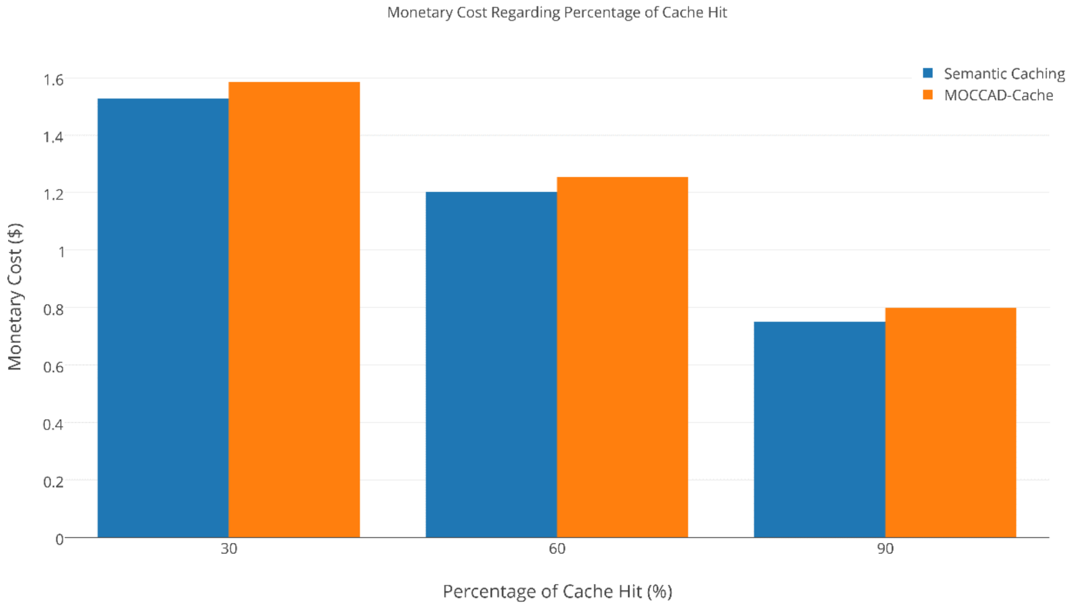


Figure 55 - Monetary Cost Regarding Cache Hit Percentage

Chapter 5: Conclusion and Future Works

In this research, we have proposed a cost-aware semantic caching algorithm and architecture called MOCCAD-Cache used within a 3-tier architecture. The goal of MOCCAD-Cache is to reduce the overhead of semantic caching when the result is included (extended hit) or partially contained (partial hit) in the cache. When a lot of computation is required to retrieve the result on the mobile device, our algorithm can choose to process the query on the cloud if it is less expensive. Indeed, 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 and if it respects the user constraints. In order to avoid additional overhead due to the computation of the estimations, our caching algorithm uses two estimation caches. One is used to store the estimations related to query processing on the mobile device and another one is used to store the estimations related to query processing on the cloud. Finally, it can process the chosen query plan, retrieve the result and replace the content of the query cache following the LRU replacement policy.

To experiment the proposed solution, a prototype has been developed on an Android device communicating with a private cloud. Thanks to the prototype's user interface, it is possible to specify constraints, to choose experimentation parameters and to build queries in order to process them. On the cloud side a 200000-tuple database has been set up on the Hadoop Framework and can be processed thanks to HiveQL queries. A pricing model has been used to determine the cost of using the cloud instances. Several set of queries have been processed on this environment each matching a given percentage of exact hit, extended hit, and partial hit. For each of those types of cache hit, we compared the performance in processing time, energy consumption and monetary cost between a query processor without cache, a query processor with a semantic cache, and a query processor with MOCCAD-Cache. Despite several environmental issues such as Wi-Fi throughput variation or processing time variations on the cloud, we have been able to make several conclusions from the experimentation results. The next section summarize those results and conclusions.

5.1 Summary of the Performance Evaluation Results

MOCCAD-Cache is a decisional semantic caching algorithm and architecture which can be used to respect the user constraints in terms of processing time, energy consumption and monetary costs, and is an improvement of semantic caching for the following reasons:

1. The MOCCAD-Cache algorithm is as efficient as the semantic caching algorithm regarding exact hit percentage in terms of time, energy and money. This is allowed by the estimation caches which prevent the many additional computations.
2. Admitting that the cloud chooses its best query plan for the given constraints and optimization parameters, the MOCCAD-Cache algorithm is more efficient than the semantic caching algorithm in terms of time regarding extended hit. The energy remains globally the same due to the result size of the queries chosen to be processed on the cloud rather than on the mobile device. However, we can observe a little overhead in terms of money since more queries are processed on the cloud.
3. The MOCCAD-Cache algorithm is equivalent to the semantic caching algorithm regarding partial hit. Indeed, the download time becomes an important criteria

since it costs processing time and energy. Becoming the bottle neck of this algorithm since the cloud performs query faster with more instances, this download time needs to be minimum for the algorithm to be efficient. Thus, processing only a part of a query on the cloud is better than processing the whole query on the cloud which makes the query plan equivalent to the one used in the semantic caching algorithm regarding partial hit. Also, money cost becomes equivalent for the MOCCAD-Cache query processor, the semantic caching query processor and even the query processor without cache.

4. Globally, our algorithm shows that MOCCAD-Cache can be used to perform faster queries within a Mobile-Cloud environment. However, the user would have to pay the price consequently.

5.2 Future Works

MOCCAD-Cache is the first decisional semantic caching algorithm and architecture within a Mobile-Cloud database system. This first step of optimization and decision making within a caching algorithm on mobile needs many expansions and opens many possibilities.

Firstly, the used optimizer has been chosen to be really simplistic to prove that only a simple optimization with semantic caching can be relevant. However, using a multi-parameter optimization to minimize time, energy and money while respecting the user constraints seems essential.

Secondly, it would be interesting to see how this algorithm behave on a public cloud due to the elaborate pricing, storage and computing model available on this platform. It would then be relevant to see how the MOCCAD-Cache behaves in the 3-tier architecture made of several heterogeneous clouds and a data owner.

Lastly, the considered operations are only selections. Thus, several other types of operation need to be considered such as projections and joins. Being able to process queries with those operations within this Mobile-Cloud environment is necessary in order to make the prototype as close as possible to a real application case.

References

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," *NIST,* vol. 53.6, p. 50, 2009.

[2] Olston et al., "A view of cloud computing," *Communications of the ACM,* pp. 50-58, 2010.

[3] N. Fernando, S. W. Loke and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems* , vol. 29, no. 1, pp. 84-106., 2013.

[4] M. e. a. Satyanarayanan, " The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE,* vol. 8, no. 4, pp. 14-23, 2009.

[5] A. Delis and N. Roussopoulos, "Performance and scalability of client-server database architectures," in *VLDB*, 1992.

[6] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava and M. Tan, "Semantic data caching and replacement," *VLDBJ,* vol. 96, pp. 330-341, 1996.

[7] A. Carroll and G. Heiser, "An Analysis of Power Consumption in a Smartphone," in *USENIX annual technical conference,* 2010.

[8] U. D. o. H. a. H. Services, "US Department of Health and Human Services," [Online]. Available: http://www.hhs.gov/ocr/privacy/. [Accessed 23 11 2014].

[9] R. Perriot, J. Pfeifer, L. d'Orazio, B. Bachelet, S. Bimonte and J. Darmont, "Cost Models for Selecting Materialized Views in Public Clouds," *International Journal of Data Warehousing and Mining,* 2014.

[10] B. Chidlovskii and U. M. Borghoff, "Semantic caching of Web queries," *VLDBJ,* vol. 9.1, pp. 2-17, 2000.

[11] B. Þ. Jónsson, M. Arinbjarnar, B. Þórsson, M. J. Franklin and D. Srivastava, "Performance and overhead of semantic cache management," *ACM Transactions on Internet Technology (TOIT),* vol. 6.3, pp. 302-331, 2006.

[12] Q. Ren and M. H. Dunham, "Using semantic caching to manage location dependent data in mobile computing," in *Proceedings of the 6th annual international conference on Mobile computing and networking*, 2000.

[13] K. C. Lee, H. V. Leong and A. Si, "Semantic Query Caching in a Mobile Environment," *ACM SIGMOBILE Mobile Computing and Communications Review,* vol. 3.2, pp. 28-36, 1999.

[14] Q. Ren, M. H. Dunham and V. Kumar, "Semantic caching and query processing," *Knowledge and Data Engineering, IEEE Transactions on,* vol. 15.1, pp. 192-210, 2003.

[15] 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," in *Emerging Technologies (ICET), 2011 7th International Conference on, IEEE*, 2011.

[16] M. Ahmad, M. Qadir and M. a. B. M. Sanaullah, "An efficient query matching algorithm for relational data semantic cache," in *Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on, IEEE,* 2009.

[17] M. Ahmad, S. Asghar, M. A. Qadir and T. Ali, "Graph based query trimming algorithm for relational data semantic cache," in *Proceedings of the International Conference on Management of Emergent Digital EcoSystems, ACM*, 2010.

[18] S. Guo, W. Sun and M. A. Weiss, "Solving satisfiability and implication problems in database systems," *ACM Transactions on Database Systems (TODS),* vol. 21, no. 2, pp. 270--293, 1996.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein and others, Introduction to algorithms, MIT press Cambridge, 2011.

[20] A. Silberschatz, H. F. Korth and S. Sudarshan, Database system concepts, McGraw-Hill New York, 1997.

[21] M. Gordon, L. Zhang, B. Tiwana, R. Dick, Z. Mao and L. Yang, *Power Tutor, A Power Monitor for Android-Based Mobile Platforms,* 2009.

[22] "Amazon Glacier," 2015. [Online]. Available: http://aws.amazon.com/glacier/. [Accessed 2015].

[23] Amazon, "Amazon S3," 2015. [Online]. Available: http://aws.amazon.com/s3/. [Accessed 2015].

[24] J. a. S. G. Dean, "MapReduce: simplified data processing on large clusters.," *Communications of the ACM,* vol. 51.1, 2008.

[25] Thusoo et al.;, "Hive: a warehousing solution over a map-reduce framework," *VLDB,* pp. 1626-1629, 2009.

[26] Olston et al., "Pig latin: a not-so-foreign language for data processing.," in *ACM SIGMOD international conference on Management of data*, 2008.

[27] Bruno et al., "Continuous cloud-scale query optimization and processing," *VLDB,* vol. 6.11, pp. 961-972, 2013.

[28] H. e. a. Kllapi, "Schedule optimization for data processing flows on the cloud," in *ACM SIGMOD International Conference on Management of data*, 2011.

[29] L. d'Orazio, "Caches adaptables et applications aux systèmes de gestion de données reparties a grande échelle," 2007.

[30] Google, "Android Developer," [Online]. Available: https://developer.android.com/index.html.