Implementation of a Novel Cache Replacement Strategy Based on Frequency & Query Cost Objectives for a Mobile Cloud Database System

Zachary Arani[#1], Christoph Kinzel[#2], Jason Arenson[#3], Chenxiao Wang [#4], Le Gruenwald [#5], Laurent d'Orazio [*6]

[#] *School of Computer Science, University of Oklahoma*
Norman, Oklahoma, USA

[1] myrrhman@ou.edu [2] christoph.c.kinzel@gmail.com [3] arensonjt@ou.edu [4] chenxiao@ou.edu

[5] ggruenwald@ou.edu

[*] *Univ Rennes, CNRS, IRISA*
Lannion, France

[6] laurent.dorazio@univ-rennes1.fr

Abstract—**We introduce the Value-Relevance Score, a new measure of the importance of cache entries in mobile-cloud database environments. We use this score to create a new cache replacement policy, the Value-Relevance Cache Replacement Policy. Using the MOCCAD mobile-could database environment, we compare user costs for this new replacement policy and the Least Recently Used cache replacement policy during query execution. We find that in the general use case, the Value-Reference method is not as effective as a Least Recently Used Implementation. We hope to refine the experiments presented here to demonstrate more precisely the effects and uses of the Value-Relevance Replacement Policy.**

I. Introduction

A mobile-cloud database environment consists of a user on a mobile device querying a database on a cloud server. Some of the data queried may be fully or partially available in the mobile device's cache storage, so that it may be desirable to run parts of the query on the cache in the interests of saving time, energy, or money. A semantic cache allows for the calculation of which parts of the query are already present in cache. However, it may not always be desirable for the user to run large amount of queries on the cache, as this could drain energy more quickly than querying the cloud. A candidate for how and where to run a specific query is referred to as a Query Execution Plan (QEP). A given QEP might involve executing the query in cache or on the cloud server, or a combination of the two. The problem of selecting a QEP given user preferences is discussed in [1]. A Specific solution to this problem is discussed in more detail in the Background/Purpose section.

In a general setting, the choice of which elements to keep in the cache in the event of cache overflow is dictated by a Cache Replacement Policy. Many well-studied cache replacement policies exist; the optimal policy depends on the specifics of the situation. One of the simplest cache replacement policies is the Least Recently Used Policy (LRU). In this model, the cache entries are maintained in a queue in order of least recent use, so that the most recently used cache element is in the back of the queue and the least recently used is at the front. When elements must be removed from the cache, they are removed from the front of the queue until the cache no longer overflows. Other models base removal on frequency [4] or some predefined cost/benefit relation [2]. In a mobile-cloud database environment, choosing which elements remain in cache is especially important. If the cache maintains a list of entries that

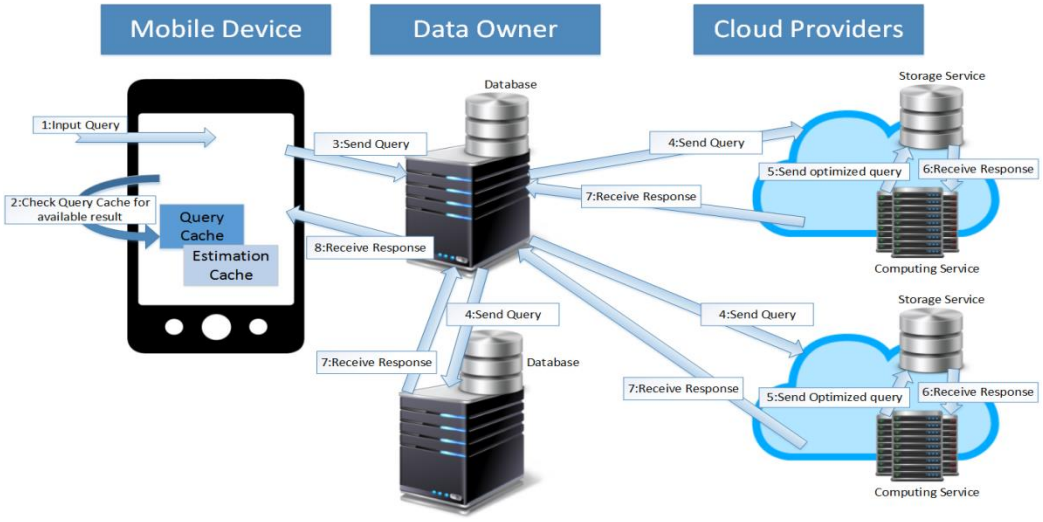

Figure 1. Mobile - Cloud Database Environment [1]

are valuable and relevant to the user, then associated costs for the user should decrease. In an attempt to accomplish this, this paper introduces a cache replacement policy based on cost/benefit (value) and frequency (relevance).

II. Background and Proposal

A. Background

This paper builds on the existing Mobile Cloud Cost-Aware Database System (MOCCAD) prototype described in [1]. The MOCCAD prototype implements a strategy for determining which QEP to execute based on user preferences and the current contents of the cache. These preferences are expressed in terms of weights the user assigns to monetary cost, energy cost, and time. Monetary cost is the amount charged by the cloud service to run the QEP, energy cost is measured in charge expended by the user's phone to run the QEP, and time is the time the QEP takes to complete. A user whose only priority is getting results quickly might assign time a weight of 1 and energy and monetary costs weights of 0, while a user having little charge remaining on their phone might assign energy cost a higher weight than the other costs.

The costs associated to a given QEP are estimated by the MOCCAD prototype. In general, running queries on the cache decreases monetary and temporal costs and increases energy costs, while the opposite holds true for running queries on the cloud. To decide which of the potential QEPs is selected, the MOCCAD implements the Normalized Weighted Sum Algorithm (NWSA). In the NWSA, each QEP is associated with a score, computed as

$$S_i = \sum_{j=1}^{n} w_j \frac{a_{ij}}{m_j}$$

(1) NWSA QEP score calculation

Here S_i is the score for the i^{th} QEP, w_j is the user-provided weight associated with cost j, m_j is a normalization constant, and a_{ij} is the computed value of cost j for plan i. In this case, there are only three costs: time, money, and energy, so that j ranges from 1 to 3; but the NWSA is applicable to arbitrary n. m_j is the maximum acceptable value for the given dimension. For instance, if a query should take at most 5 seconds, $m_j$would be set to 5. The QEP score synthesizes user preferences and estimated costs, so that it represents a single user-specific cost for the QEP.

The QEP with minimum score is selected by the MOCCAD prototype. The authors of [1] demonstrate improved quality of QEP selections using this score method over single-optimization strategies.

B. Integration of NWSA into MOCCAD Semantic Cache

Our purpose is to explore the effects of alternative cache replacement strategies on the performance of the MOCCAD prototype. In [1], the prototype implemented an LRU cache replacement strategy. By choosing a more sophisticated strategy tailored to query history and user preference, the contents of the cache may be made more relevant and valuable to the user. As this occurs, QEP scores should decrease as more common and expensive queries have the option of being run on the cache. We propose a cache replacement policy based on a score involving relevance and value. In this policy, the elements of the cache are kept in sorted order by this constantly updating score throughout execution of the MOCCAD prototype. In the event of cache overflow, the element with the smallest score is removed from the cache; this process is repeated until the cache overflow is resolved. The cache score for cache entry k is computed as

$$C_k = \frac{f_k S_k}{L_k}$$

(2) Value-Relevance Score for cache entries.

Here f_k is a measure of the frequency with which the contents of cache entry k has intersected with subsequent queries. Many frequency measures are possible; for our experiments, we chose to reset all frequencies to 0 every p queries, where the value of p was altered over various experiments; see the results section for the values of p tested. This frequency measure reflects how often the data has been used recently; hence it should correlate with relevance. Its limitations are discussed in future work.

S_k is the QEP score of the query corresponding to cache entry k. Since this QEP score represents a user-specific cost, a cache entry with higher QEP score is more valuable to the user; hence QEP score should correlate with value. In our implementation, we assume the user's preferences do not change during the execution of the MOCCAD prototype. This means QEP scores do not need to be recalculated once elements are placed in the cache. This greatly reduces the number of recalculations needed once elements are in the cache. This is arguably not a realistic assumption; see the future work section for more discussion.

L_k is the size of cache entry k. Since larger entries prevent more data from being stored in the cache, we divide by L_k to encourage smaller entries that are still valuable and relevant. There are many candidates for size; in our implementation, we chose a linear measure of size.

In general, we call scores of the form $\frac{fS}{L}$, where f is a measure of frequency, S is the QEP score associated with the cache entry, and L is a measure of cache entry size, a *Value-Relevance Score*. Cache replacement policies induced by such a score are *Value-Relevance cache replacement policies*.

By using a Value-Relevance Replacement policy instead of the LRU cache replacement policy, we hope elements of the cache will be more valuable and relevant to the user. As a result, QEP scores for new queries should be lower once the cache is filled for this score-based replacement policy compared to the original LRU cache replacement policy. As lower QEP scores represent lower user-specific costs, this would represent an improvement in the performance of the MOCCAD prototype.

B.A.1 *Implementation of the Value-Relevance Cache Replacement Method*

A min-heap implementation of a priority queue is used for the query cache. When an entry is added, the cache first checks if the entry already exists within the cache, if so, it runs an update procedure, which updates the score of each cache entry. If the entry does not exist in the cache, it then checks to see if other entries need to be removed in order to fit the new entry. If so, it iteratively deletes the lowest scoring entries until there is enough space to fit the new entry. Then, the entry is added to the cache, updating each score.

III. DATA ANALYSIS

A. *Experimentation Background*

For general experimentation, several SQL tables were created using *Apache Hive*. A set of 25,000 tuples were programmatically generated with the format shown in Table 2. Five tables, with increasing tuple sizes (5,000 to 25,000, incrementing by 5,000), were populated, meaning table five contained each of tables one, two, and so on, along with 5,000 other entries.

Queries were also programmatically generated. Given a table and a set amount, a list of queries would be generated that were guaranteed to be contained within the given table. Each query was a simple *SELECT* command, containing only one attribute, as seen in Table 3. A set of 250 queries were generated and used for all testing. For instance, the query workload experiment used different portions of the generated sample: a test of fifty queries used only the first fifty queries generated, a test of one hundred used the first hundred, and so on.

B. *Default Parameter Values*

Following is a list of the default parameter values. These are the values of each parameter in each experiment unless explicitly stated otherwise.

Query Cache Segment Amount	20
Query Cache Size Limit	10 MB
Cloud/Mobile Estimation Cache Segment Amount	INF
Cloud/Mobile Estimation Cache Size Limit	INF
Relation Size	25,000
Queries Tested	250
NWSA Weights	.33
Number of Instances Simulated on Cloud	5
Frequency Locality	40 Cache References

Table 1: Static Parameter Values

C. *Frequency Locality*

As defined in (2), the frequency f_k will reset to 0 for all entries in the cache after p queries. This value p defines a locality for frequency that is very useful in maintaining relevancy. In this experiment, we tested several locality values, ranging from ten queries to ninety queries, to measure the locality's effect on cost, time, and energy. As experimentation shows, there is an ideal range for frequency locality that coincides to roughly twice that of the cache's segmentation limit.

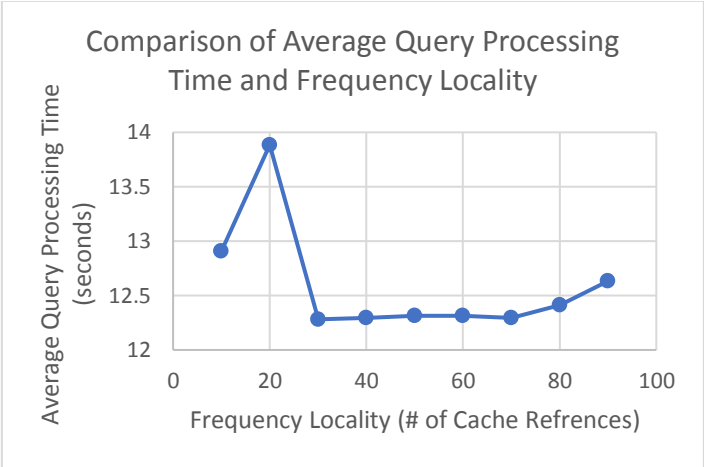

Figure 2: Comparison of Average Processing Time and Frequency Locality

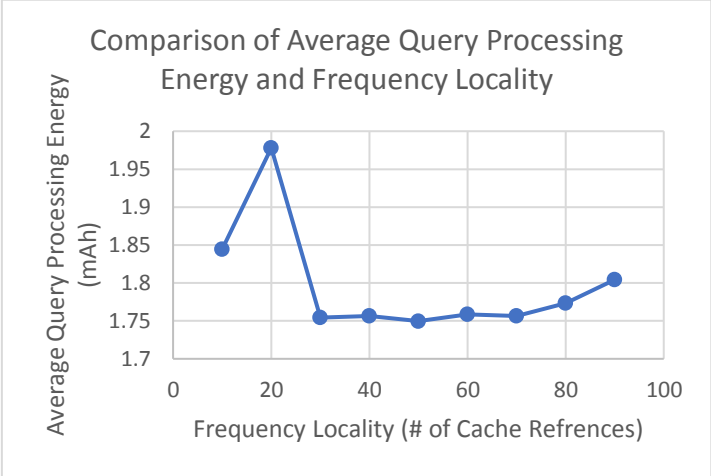

Figure 3: Comparison of Average Query Processing Energy and Frequency Locality

Name	noteid	patientfirstname	patientlastname	doctorfirstname	doctorlastname	*description*	*P_date_time*	*heartrate*
Type	Integer	String	String	String	String	String	String	String
Valid Range	(0, INF)	A-Z	A-Z	A-Z	A-Z	A-Z	1940-2014 00:00-23:59	(15,185)
Example	25	Zachary	Johnson	Christoph	Wang	Endoscopy	1985-08-02 22:51:00	85

Table 2: Relation Specification for Generated Tuples

Attribute	noteid	patientfirstname	patientlastname	doctorfirstname	doctorlastname	description	p_date_time	heartrate
Example	*SELECT * FROM table WHERE noteid < 25*	*SELECT * FROM table WHERE patientfirstname = 'Zachary'*	*SELECT * FROM table WHERE patientlastname = 'Johnson'*	*SELECT * FROM table WHERE doctorfirstname = 'Christoph'*	*SELECT * FROM table WHERE doctorlastname = 'Wang'*	*SELECT * FROM table WHERE description = 'Endoscopy'*	*SELECT * FROM p_date_time WHERE substr(p_date_time,0,10) >= '2003-09-23'* *SELECT * FROM p_date_time WHERE substr(p_date_time,12) = '05:22:00'*	*SELECT * FROM table WHERE heartrate = 85*

Table 3: Examples of Generated Queries

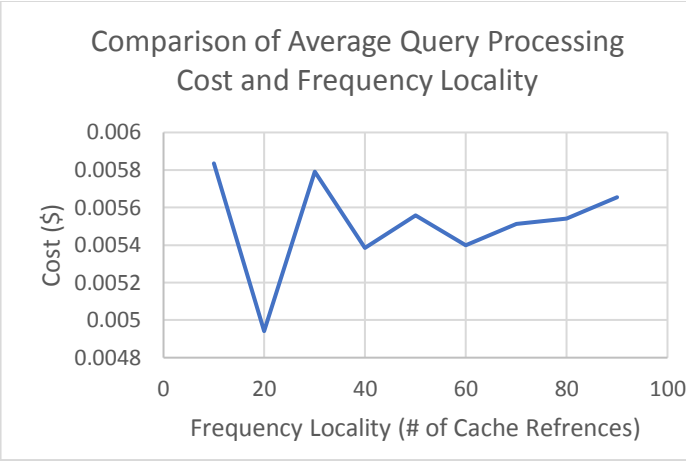

Figure 4: Comparison of Average Query Processing Time and Frequency Locality

These results are consistent with our expectations. It is predictable that a small locality would lead to a lack of relevant data in the cache, causing constant unnecessary replacement. We also assumed that having an overtly large locality would also be inefficient, as a large portion of the cache would contain irrelevant data, thereby wasting precious space.

As the results show, time and energy expenses grow quickly as the locality reaches the segmentation size of the cache. However, when the locality reaches roughly twice the size of the cache's segmentation limit, both time and energy are optimized. Conversely, the average cost of a query decreases as the energy and time required for processing increase. As these three objectives are related, it is expected that cost would react to the changes in energy and time.

In a simple case, it is clear that using a locality approximately twice the size of the query's segmentation limit leads to the most optimal results. In the following experiments, a locality of 40 cache references for the rest of our experimentation.

D. Effects of Relation Size on Caching Efficiency

After establishing an optimal frequency locality, our next experiment compared the effects of relation size (i.e., the number of entries in a database) on the performance of the reference-value cache replacement method. Relations sizes ranging from 5,000 to 25,000 were tested. For comparison purposes, the Least Recently Used (LRU) cache replacement strategy was also tested.

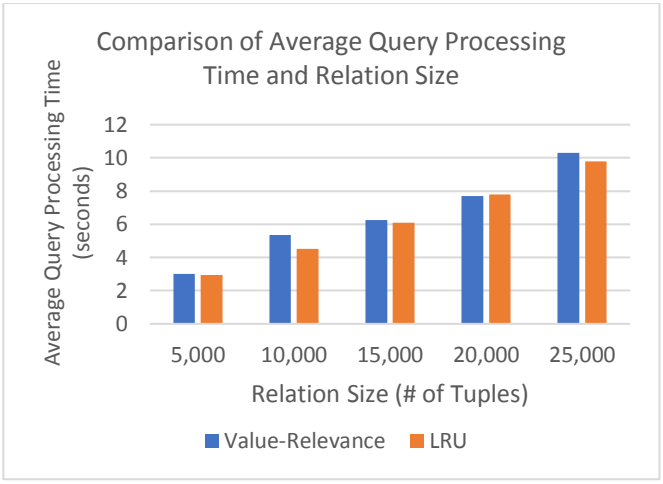

Figure 5: Comparison of Average Query Processing Time between Value Relevance and LRU Cache Replacement Policies

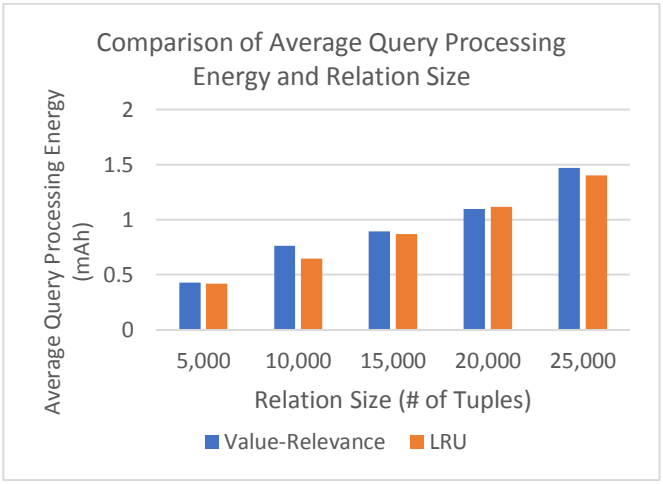

Figure 6: Comparison of Average Query Processing Energy between Value Relevance and LRU Cache Replacement Policies

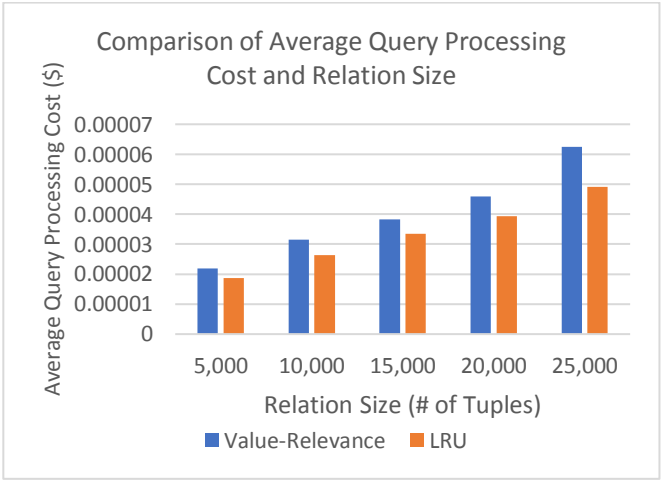

Figure 7: Comparison of Average Query Processing Cost between Value Relevance and LRU Cache Replacement Policies

LRU generally outperforms Value-Relevance in cost, energy, and time efficiency for every relation size

tested, despite using what is seemingly the most optimal frequency locality. It should be noted, however, that the differences in performance between the two are generally minute and, in a few cases the Value-Relevance method performs better than the LRU replacement strategy.

E. Effects of Query Workload on Caching Efficiency

In a similar experiment, we tested how increasingly large query sets affected the performance of the Value-Relevance replacement method and, by comparison, the Least Recently Used replacement method.

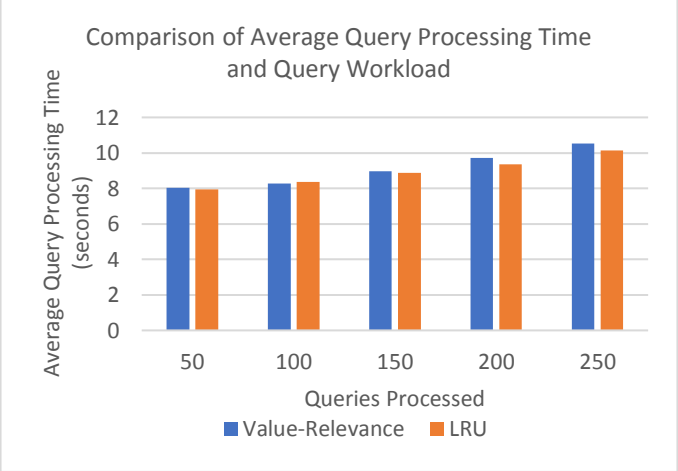

Figure 5: Comparison of Average Query Processing Time and Query Workload between Value-Relevance and Least Recently Used Cache Replacement Strategies

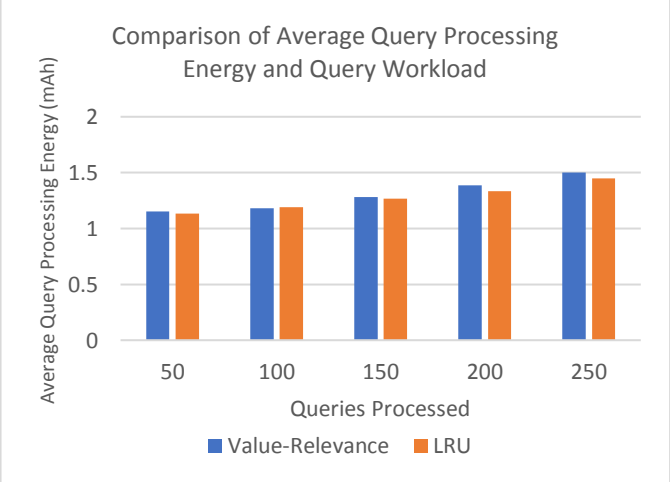

Figure 5: Comparison of Average Query Processing Energy and Query Workload between Value-Relevance and Least Recently Used Cache Replacement Strategies

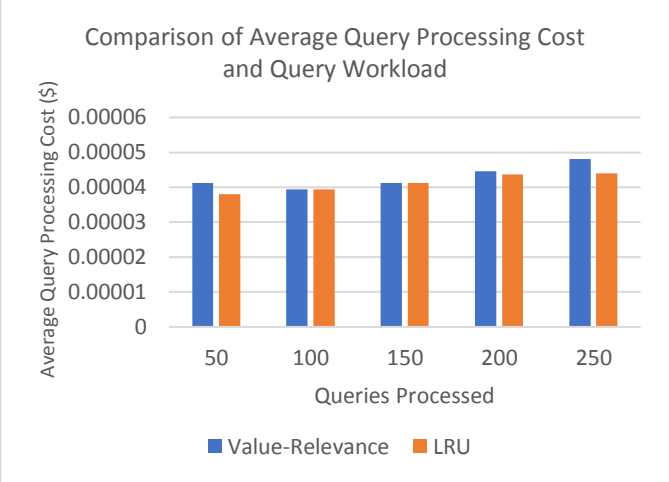

Figure 5: Comparison of Average Query Processing Cost and Query Workload between Value-Relevance and Least Recently Used Cache Replacement Strategies

As query workload increases, the Least Recently Used algorithm tends to be slightly more efficient than the Value-Relevance replacement method in terms of cost, energy, and time (when weighted equally). This difference is very minimal but appears to increase as the workload grows.

IV. Conclusion

In a general use case, the Value-Relevance replacement method simply does not hold up to the more simplistic Least Recently Used strategy. Even though there are potential advantages in using Value-Relevance in certain situations, the general performance of the replacement method does not justify amount of overhead required to maintain the cache.

V. Future Work

The experiments discussed here are incredibly general and arguably have an abundance of unrealistic assumptions about the user and the information in the database. Each assumption represents a potential avenue for future work.

In our implementation of the Value-Relevance Score, we assume the preferences of the user do not change, so that effective QEP scores, S_k in (2), do not change for cache elements. In practical applications, it's likely the user's preferences do not remain constant. Hence, it may be desirable to recalculate QEP scores when new user preferences are provided. The simplest option is to recalculate QEP score based on these new weights. More sophisticated methods include averaging previous user preferences to produce a set of weights that takes into account historical usage, possibly weighting more recent preferences more. The prediction of user weight preferences is also a potential application of machine learning. Of course, even these concepts assume that the environment of the MOCCAD prototype remains constant, so that the costs a_{ij} in

(1) remain constant. If this assumption is not justified, a recalculation scheme for these costs must be implemented as well.

Many measures of size are possible for L_k in (2). Although we implemented a linear measure, it's possible a quadratic measure of size may be more appropriate, as multiplying the size of a cache entry by a given factor increase its value (QEP score) and relevance (frequency) by similar factors.

Our measure of frequency, f_k, in (2) was chosen for its ease of implementation and low overhead. A more sophisticated measure of a cache entry's hit frequency might calculate frequency based on the last p queries instead of resetting every p. Although this clearly represents a better notion of frequency than our reset-based measure, it requires more calculations and a new data structure, which may introduce significant added energy and time costs.

Our assumptions in data generation also limit the applicability of these results. Both the database and the queries were randomly generated. This makes the relevance portion of the Value-Relevance Score, in a sense, useless in this application. If more query patterns were tested (e.g. % Cache Hits, % Cache Extended Hits, etc.), the Value-Relevance Score could better reflect the relative importance of cache entries.

VI. Acknowledgement

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

VII. References

[1] Helff, Florian, Le Gruenwald, and Laurent d'Orazio, "Weighted Sum Model for Multi-Objective Query Optimization for Mobile-Cloud Database Environments." EDBT/ICDT Workshops. 2016.

[2] Jean-Chrysostome Bolot, Philipp Hoschka, "Performance engineering of the World Wide Web: Application to dimensioning and cache design." Proceedings of the Fifth International World Wide Web Conference 6-10 May 1996, vol. 28, no. 7-11. pp. 1398-1405, 1996.

[3] M. Perrin, Time-, Energy-, and Monetary Cost-Aware Cache Design for a Mobile Cloud Database System," Master's Thesis, University of Oklahoma, School of Computer Science, May 2015.

[4] Nabil Kouici, Dennis Conan, and Guy Bernard, "Caching components for disconnection management in mobile environments." International Conference on cooperative Information Systems, pages 1322-1339, 2004.

[5] MOQP: Immanuel Trummer, Christoph Koch: Multi-objective parametric query optimization. VLDB vol. 26, no. 1: 107-124 (2017)

[6] Semantic caching: Qun Ren, Margaret H. Dunham, Vijay Kumar: Semantic Caching and Query Processing. IEEE Trans. Knowl. Data Eng, vol. 15 no. 1, pp. 192-210, 2003.