A Scored Semantic Cache Replacement Strategy for Mobile Cloud Database Systems

Zachary Arani, Drake Chapman, Chenxiao Wang, Le Gruenwald, Laurent d'Orazio, Taras Basiuk

A Scored Semantic Cache Replacement Strategy for Mobile Cloud Database Systems

Zachary Arani[1], Drake Chapman[1], Chenxiao Wang[1], Le Gruenwald[1], Laurent d'Orazio[2], and Taras Basiuk[1]

[1] The University of Oklahoma, Norman OK 73019, USA
{myrrhman, christopher.d.chapman, chenxiao, ggruenwald, taras.basiuk}@ou.edu
[2] Univ Rennes, CNRS, IRISA Lannion, France
laurent.dorazio@univ-rennes1.fr

Abstract. Current mobile cloud database systems are widespread and require special considerations for mobile devices. Although many systems rely on numerous metrics for use and optimization, few systems leverage metrics for decisional cache replacement on the mobile device. In this paper we introduce the Lowest Scored Replacement (LSR) policy—a novel cache replacement policy based on a predefined score which leverages contextual mobile data and user preferences for decisional replacement. We show an implementation of the policy using our previously proposed MOCCAD-Cache as our decisional semantic cache and our Normalized Weighted Sum Algorithm (NWSA) as a score basis. Our score normalization is based on the factors of query response time, energy spent on mobile device, and monetary cost to be paid to a cloud provider. We then demonstrate a relevant scenario for LSR, where it excels in comparison to the Least Recently Used (LRU) and Least Frequently Used (LFU) cache replacement policies.

Keywords: Big Data · Cloud Computing · Caching

1 Introduction

Since a cache has limited space, it is important to use replacement policies which keep relevant data on a mobile device. In a mobile cloud database system, querying the cloud can often be an expensive operation in regards to time, money paid to a cloud provider, and mobile device energy. For this reason, leveraging a cache grants large boosts in efficiency. The rudimentary Least Recently Used (LRU) policy—which discards the least recently accessed entry when filled—is often implemented in caches. The similar Least Frequently Used (LFU) policy—which replaces the least frequently used entry when full—is also commonly implemented; however, LRU and LFU are not always the most efficient policies within the context of a relational database system [3, 8, 12, 13]. Instead, many Database Management Systems (DBMS) implement specific replacement policies that cater to the system's needs.

This paper seeks to describe a cache replacement policy for mobile cloud database systems that utilizes decisional semantic caching. We propose the Lowest Scored Replacement policy (LSR), which takes cache relevancy and mobile constraints into account while maintaining a LRU-like overhead. LSR partitions cache events into point scoring categories and utilizes a predefined score based on decisional semantic caching and the Normalized Weighted Sum Algorithm. LSR defines semantic relevancy within the cache while also taking into account user preferences on saving time, energy on mobile device, and cost paid to cloud providers. We then show an implementation of LSR using our existing decisional semantic cache system and demonstrate a relevant scenario for the algorithm. We find from our experiment that there exist scenarios where LSR significantly outperforms the common LRU and LFU cache replacement policies in the mobile cloud database environment.

2 Related Work

Device status and metadata is imperative in mobile cloud database decision making. Metrics such as current battery life, location, and connectivity quality may be leveraged to contextualize computational tasks. Mobile devices are, by their very nature, constrained through their short-term battery life and variable connection to wireless networks. Several replacement policies have been developed to address these issues [1]. These policies make use of metrics such as location, battery life, and on-device data size [2, 7, 15]. Other caching systems, such as semantic caching [11], have also been used to address constraints in a mobile cloud database environment. In our previous work we developed the decisional semantic MOCCAD-Cache [10] as well as the Normalized Weighted Sum Algorithm (NWSA) [4] to better meet constraints on a mobile device. However, our previous work did not propose any solution for a cache replacement policy.

One cache replacement approach that bears some similarities to ours is frequency-based. This method considers each cache entry's access frequency when performing decisional replacement. Examples are found in [6] where three different methods are proposed—each with its own contextual merits. These policies—*The mean scheme*, *The window scheme*, and *The exponentially weighted moving average scheme*—bear some resemblance to ours since they take into account how recently a cache entry was accessed; however, these cache replacement strategies do not take into account semantic information or mobile constraints such as the device's battery life.

There exist other policies which take data size and cloud retrieval cost into account. One example is the SAIU (*Stretch Access-rate Inverse Update-frequency*) replacement policy proposed in [14], as well as a modified version in [5]. The SAIU defines a gain function based on access rate, update frequency, data size, data retrieval time, and (in the latter paper) consistency. The policy then uses this weight to determine replacement. A version proposed in [9] uses a more generalized cost function. By taking the various costs of each query into account, these methods for cache replacement share some similarity with ours; however, these methods do not factor the benefits of semantic caching into their replacement process.

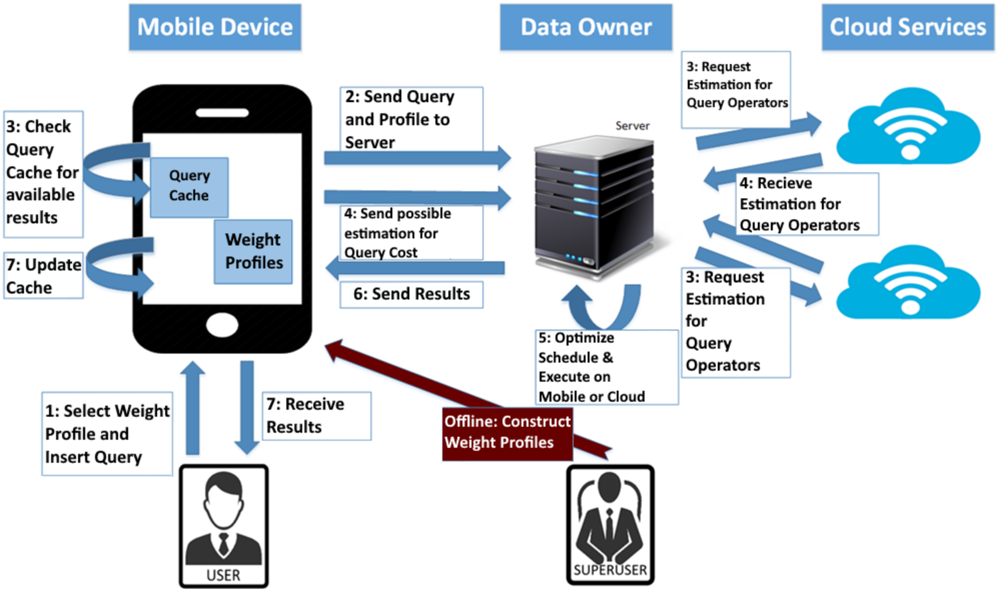

Mobile Device
Data Owner
Cloud Services
3: Check Query Cache for available results
Query Cache
Weight Profiles
7: Update Cache
2: Send Query and Profile to Server
4: Send possible estimation for Query Cost
6: Send Results
Server
3: Request Estimation for Query Operators
4: Recieve Estimation for Query Operators
3: Request Estimation for Query Operators
5: Optimize Schedule & Execute on Mobile or Cloud
Offline: Construct Weight Profiles
1: Select Weight Profile and Insert Query
7: Receive Results
USER
SUPERUSER

Fig. 1. Our Mobile Cloud Database Environment

3 Our Mobile Cloud Database System

Figure 1 details our system architecture and illustrates the possible workflows for query execution. In our mobile cloud database environment, a user sends queries and constraints (in the form of a weight profile) from the mobile device for execution. User constraints are defined by their interest in conserving time, mobile device energy, or monetary cost paid to their cloud provider. Our system decides how to best execute queries with respect to user constraints: either through the cloud or from a local cache if possible. The user's query and weight profile is sent through some form of infrastructure (the data owner) to cloud services for query execution estimations. The data owner then uses these estimations to decisionally optimize query execution on some combination of the mobile device and the cloud.

3.1 MOCCAD-Cache

Our previously proposed MOCCAD-Cache introduces the concept of decisional semantic caching [10] to solve issues with semantic caching in mobile contexts.

In a semantic caching system, both data and the semantic description of that data are provided to the cache [11]. For instance, the result of a query will be stored along with what relations, attributes, and predicates the query consisted of. This means new queries can be compared against semantic descriptions of stored queries before execution.

Many systems only detect a cache hit event if both the input query and the cache query match exactly. Semantic caching instead allows recognition of overlaps in semantic descriptions. If an input query has overlap in the relations, attributes, and predicates of a cached query, it will be recognized as either a *partial hit* (some data in the cache) or an *extended hit* (all data in cache). If the query semantics match exactly or do not match at all, a cache hit or miss is detected. If the parser detects some semantic overlaps, a query trimmer is used to create two new queries for execution. For example, if only part of the input query is stored in the cache—a partial hit event—the query trimmer will transform the input query into a mobile device query (probe query) and a cloud query (remainder query) for execution. If the input query's semantic description is fully overlapped with cached semantics, it is recognized as an extended hit event; the query is then executed on the mobile device using cached data. Semantic caching is useful in many cases, especially on a device with limitations such as a mobile phone. However, there are some instances where local query trimming and execution is considerably worse than a simple cloud execution. After trimming the query, the MOCCAD-Cache runs estimations to decide whether to execute the query on the cloud or mobile device—if possible. Despite needing to go through an entirely new estimation phase, this system can improve query processing time (albeit with the general caveat of increased monetary cost).

3.2 Normalized Weighted Sum Algorithm

MOCCAD-Cache may outperform semantic caching in terms of time, but there are many scenarios where time is not the only imperative metric. Our later proposed Normalized Weighted Sum Algorithm (NWSA) addresses this issue by taking into account arbitrary metrics and normalizing them to calculate a score [4]. The NWSA calculates a score based in part on the user's interest in saving time, energy, or monetary cost[3]. These constraints are evaluated against query execution plans (QEPs) to decide a QEP that best respects the user's desires.

$$\begin{gathered} QEP_1 : \{M = \$0.080; T = 0.5s; E = 0.012mA\} \\ QEP_2 : \{M = \$0.050; T = 3.0s; E = 0.300mA\} \\ QEP_3 : \{M = \$0.055; T = 0.6s; E = 0.013mA\} \end{gathered}$$

Fig. 2. Example Query Execution Plans (QEPs) for a Sample Query

[3] Each parameter is given on a scale from 0 to 1, where the sum of all parameters must total 1

$$QEP_{score} = \min_{i} \sum_{j=1}^{n} w_j \frac{a_{ij}}{m_j}$$

Fig. 3. Equation for Finding the Best QEP Score Using the Normalized Weighted Sum Algorithm

$$w_j = \frac{uw_j * ew_j}{\sum uw * ew}$$

Fig. 4. Equation for Composite Normalized Weight Factor

The MOCCAD-Cache estimates the efficiency of executing a query on the cloud versus the mobile device. If the query is performed on the cloud, there are several QEPs that can be executed–each with its own costs. Some example QEPs for a given query are shown in Figure 2. The NWSA takes each QEP and scores how it respects user constraints. The lowest scoring QEP signifies the most efficient execution respecting user constraints. For instance, a user giving priority to time and monetary cost would result in QEP_3 from Figure 2 being executed, as it respects those two constraints the most.

The methodology of QEP scoring is shown in Figure 3. The NWSA looks at each QEP and scores it based on three factors. a_{ij} is the ith QEP's estimated cost for the jth constraint (money, time, energy). m_j is the maximum accepted value for the jth constraint. Any QEP with a constraint value higher than m_j is not considered for best score. Figure 4 describes w_j, a composite normalized weight factor derived from the user constraints (uw) as well as a device's environmental factors such as battery life or network connectivity (ew).

In summary, the MOCCAD-Cache with the NWSA dictates the structure of cache entries and how to handle new cache events, but it does not detail any methods of cache replacement and may assume an infinite cache size. In section 4 we propose a novel replacement policy leveraging data calculated by the MOCCAD-Cache using NWSA to efficiently replace entries while respecting constraints.

4 The Lowest Scored Replacement (LSR) Policy

This section explores our proposed cache replacement policy and its implementation. The Lowest Scored Replacement Policy (LSR) utilizes the QEP score calculated by the NWSA as well as cache events defined by the MOCCAD-Cache. A modified QEP score along with MOCCAD-Cache cache events score each cache entry for decisional replacement.

4.1 LSR Score

The initial LSR score of a cache entry is based on the QEP score for each query. Higher QEP scores indicate that a given query is more difficult to retrieve from the cloud, and therefore may be more valuable to keep in the cache. For the sake of precision and arithmetic simplicity, we keep baseline LSR scores close to the magnitude of 1. This requires the given QEP score to be multiplied by a scaling factor before being considered as the LSR score.

$$LSR_{Score} = QEP_{Score} * ScaleFactor$$

4.2 Scoring System

After being initialized, an entry's score is updated by cache events. The most desirable cache event—a cache hit—is rewarded a *FULL-POINT* whereas the least desirable outcome is given a *ZERO-POINT* score. Scores for cache events involving query trimming fall between these two extremes. The extended hit cache event (all relevant data in cache) is rewarded a *HALF-POINT*, as this event generally does not involve accessing the cloud. The partial hit cache event (some relevant data in cache) is rewarded a *QUARTER-POINT*, since some local data is often more desirable than a cache miss.

Since LSR rewards cache entries based on accessed data, it is worth noting its similarity to LFU. Unlike LFU, LSR takes into account the constraints of the mobile device (the QEP score) along with the query's semantic utility in the cache (MOCCAD-Cache events). This functionality is crucial for mobile cloud computing, where devices are constrained by limited resources and user requirements. In short, LSR inherently respects the constraints of mobile computing, unlike LFU.

4.3 Cache Implementation

The cache is implemented as a minimum priority queue. Entries are initialized with a given score and are updated as cache events occur; the cache entry score dictates the priority within the queue. When an entry needs to be removed, the lowest scoring entry is replaced.

5 Experimentation and Results

In this section we discuss our experiments—and the methodology behind them—as well as their results. We have conducted an experiment to compare the performance of our proposed algorithm, LSR, against the ubiquitous LRU and LFU policies. We compare these replacement policies in terms of the monetary cost, query response time, and energy consumption.

Table 1. Summary of Static Experiment Parameters

Static Parameters	Value	Reference
LG V10 Memory	2GB	Kernel
LG V10 SoC CPU Active Mode Frequency	1.728 GHz	Kernel
LG V10 SoC CPU Active Mode Current	157.44 mA	Power profile
LG V10 SoC CPU Idle Mode Frequency	0	Kernel
LG V10 SoC CPU Idle Mode Current	16.4	Power profile
LG V10 SoC Wi-FI Network Low Current	0.1 mA	Power profile
LG V10 SoC Wi-FI Network High Current	60 mA	Power profile
LG V10 Battery Capacity	3000 mAh	Power profile
LG V10 Average Bandwidth Up	8.47 Mbps	Google Speedtest
LG V10 Average Bandwidth Down	12.9 Mbps	Google Speedtest
Cloud Node Memory	16 GB	Kernel
Cloud CPU	Intel i7-8750H @ 2.20 GHz	Kernel
Cloud Node Disk	512 GB Samsung 970 EVO NVMe PCIe M.2-2280 SSD	Kernel
Cloud Node Average Bandwidth Up	3.28 Mbps	Google Speedtest
Cloud Node Average Bandwidth Down	28.0 Mbps	Google Speedtest
Query Cache Maximum Size	100 MB	
Query Cache Maximum Entries	10	
Query Set Size	20 queries	
Dataset	TPC-H Benchmark	TPC
Number of Relations	8	TPC
Database Size	2 GB	
$ScaleFactor$	10^{12}	QEP Score Magnitude

5.1 Experimentation Hardware and Software

In order to simulate a cloud environment, a single node was set up using the Hadoop framework and data warehouse infrastructure. Apache Hive was used as the database system along with MySql for storing metadata. The node ran the Arch Linux operating system and featured an Intel Core i7-8750H CPU running at 2.20GHz as well as 16GB of RAM and a 512 GB Samsung 970 EVO NVMe PCIe M.2-2280 SSD. A RESTful java web servlet running on Apache Tomcat 8.5 was used to access the cloud infrastructure from mobile devices. This servlet is able to retrieve tuples, query cost estimations, and relational metadata from the hive server.[4] The mobile device ran a development branch of the Java based MOCCAD-Cache Android prototype that features an LSR implementation and other small improvements.[5]

The mobile device used for testing was a LG V10 Android device[6], which featured a hexa-core Qualcomm MSM8992 Snapdragon 808, 2 GB of RAM, and a battery capacity of 3000 mAh. Table 1 summarizes the experiment parameters.

5.2 Example Scenario

In this section, we outline a relevant scenario for LSR. We envision a business worker using a mobile device to access company information while away from the office–where productivity may be affected by mobile constraints. The worker spends time focusing on one business context before switching to another. They may execute several queries all related to one product, customer, or region before suddenly switching to a different one. This means that variations of a complex query will be executed several times in sequence before completely unrelated queries are executed. The worker may then return to the original context they were working in later.

In this scenario, there is a full cache with an entry that is semantically useful but is difficult to retrieve from the cloud. If the entry has not been used for a short period (a context switch), LRU and LFU will remove it. These policies remove the useful entry because they do not respect semantic utility or mobile constraints but instead only respect recent accesses. If a query from the original context is then executed, it will be very expensive in terms of time, money, and energy for the LRU and LFU users. Unlike LRU or LFU, LSR will respect the constraints of the mobile cloud database environment and retain the entry for continued use.

[4] The source code for the cloud web service can be found at https://github.com/ZachArani/CloudWebService

[5] The source code of MOCCAD-Cache and the NWSA can be found at http://cs.ou.edu/ database/MOCCAD/index.php. this experiment was conducted on the 'dev' branch.

[6] Model *LG-H900*

5.3 Experimentation Methodology

In order to simulate the database environment of a business worker, we generated a 2GB database based on the TPC-H model, which is structured to simulate business data.[7]

Before running the experiment, a series of ten *warmup queries* were executed to fill the cache with data prior to the experiment. These queries simulate previous contexts unrelated to the experimental ones in order to encourage cache replacement. We then ran twenty queries for this experiment, starting with a costly query:

> *SELECT DISTINCT l_shipdate FROM lineitem WHERE l_linestatus = 'O';*

After this, we ran several semantic hits (extended and partial) in the same context before switching to unrelated queries of a different context. After several queries are run in this context, the original query and semantic hits were then run again near the end of the workload. The experiment measured total execution time, energy spent on mobile device, and estimated cost paid to the cloud provider. The experiment was run three times, with the results being averaged. The MOCCAD-Cache prototype's user preference weights for money, energy, and time were all set to an equal one-third amount.

5.4 Results

Figures 5, 6, and 7 detail the results of our experimentation. As we expected, LSR significantly outperformed LRU and LFU in the business scenario. LSR performed over twice as fast, cheap, and energy efficient when compared to LRU. LFU managed to be somewhat competitive in cost, but still was eclipsed by LSR in speed and energy efficiency. In terms of all three metrics, LSR was clearly cheaper, faster, and more efficient in the mobile cloud database environment. Our policy used valuable metrics to recognize the utility of data as well as respect the constraints of the mobile device. Even though LFU and LRU may have only needed to run a handful of additional queries on the cloud—the re-execution of large or costly queries will not respect the constraints of a mobile device. These costs may be greatly exacerbated depending on the device's particular context. LSR, by comparison, leveraged user constraints and decisional semantic cache events to intelligently retain valuable data locally.

6 Conclusions

Mobile cloud database systems have become ubiquitous in recent memory. Several advancements have been made in the field, such as our previously proposed

[7] *hive-testbench* by HortonWorks was used for database creation. It can be accessed at https://github.com/hortonworks/hive-testbench

Fig. 5. Scenario Monetary Cost Performance

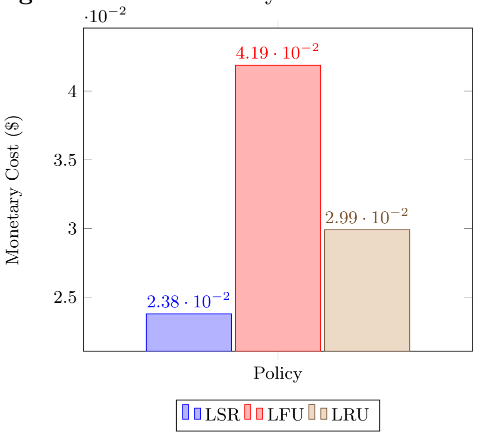


Fig. 6. Scenario Time Performance

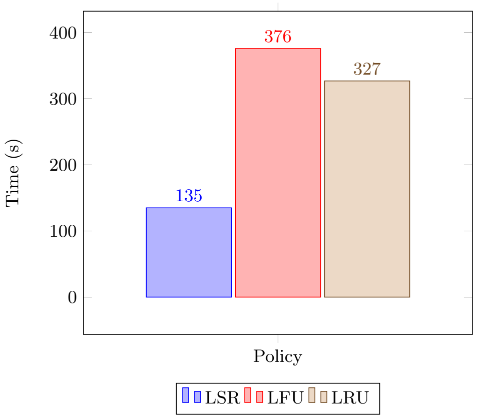
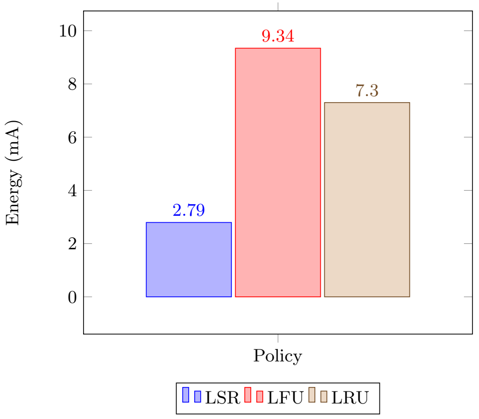


Fig. 7. Scenario Energy Performance



decisional semantic MOCCAD-Cache as well as the Normalized Weighted Sum Algorithm. LSR attempts to improve on existing cache replacement strategies in mobile cloud database systems by accounting for mobile device constraints and semantic utility. When combined with the MOCCAD-Cache and the Normalized Weighted Sum Algorithm, there exist scenarios where LSR significantly outperforms LFU and LRU in a mobile cloud database environment.

Although LSR's initial experiments are promising, future work is needed to better determine its use and applicability in the mobile cloud database setting. Experiments could be run to see how LSR compares against other common non-LRU based cache replacement policies. Experiments that vary the size of the cache would prove insightful on how useful LSR would be in other caching environments. On top of this, varying user preference weights for time, money, and energy for NWSA scoring may provide interesting results. Additional workloads and scenarios must be analyzed to investigate where LSR is most applicable to real world applications. Finally, implementing the LSR policy in non-mobile cloud database systems may also yield promising results in testing the policy's utility in other areas.

7 Acknowledgments

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

References

1. Barbará, D., Imieliński, T.: Sleepers and workaholics: Caching strategies in mobile environments (extended version). The VLDB Journal **4**(4), 567–602 (Oct 1995), http://dl.acm.org/citation.cfm?id=615232.615236
2. Chand, N., Joshi, R.C., Misra, M.: Cooperative caching strategy in mobile ad hoc networks based on clusters. Wireless Personal Communications **43**(1), 41–63 (Oct 2007). https://doi.org/10.1007/s11277-006-9238-z, https://doi.org/10.1007/s11277-006-9238-z
3. Chou, H.T., DeWitt, D.J.: An evaluation of buffer management strategies for relational database systems. In: Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11. pp. 127–141. VLDB '85, VLDB Endowment (1985), http://dl.acm.org/citation.cfm?id=1286760.1286772
4. Helff, F., Gruenwald, L., d'Orazio, L.: Weighted sum model for multi-objective query optimization for mobile-cloud database environments. In: EDBT/ICDT Workshops (2016)
5. Jianliang Xu, Qinglong Hu, Wang-Chien Lee, Dik Lun Lee: Performance evaluation of an optimal cache replacement policy for wireless data dissemination. IEEE Transactions on Knowledge and Data Engineering **16**(1), 125–139 (Jan 2004). https://doi.org/10.1109/TKDE.2004.1264827
6. Leong, H.V., Si, A.: On adaptive caching in mobile databases. In: Proceedings of the 1997 ACM Symposium on Applied Computing. pp. 302–309. SAC '97, ACM, New York, NY, USA (1997). https://doi.org/10.1145/331697.331760, http://doi.acm.org/10.1145/331697.331760

7. Li, W., Chan, E., Chen, D.: Energy-efficient cache replacement policies for cooperative caching in mobile ad hoc network. In: 2007 IEEE Wireless Communications and Networking Conference. pp. 3347–3352 (March 2007). https://doi.org/10.1109/WCNC.2007.616
8. Li, Z., Jin, P., Su, X., Cui, K., Yue, L.: Ccf-lru: a new buffer replacement algorithm for flash memory. IEEE Transactions on Consumer Electronics **55**(3), 1351–1359 (August 2009). https://doi.org/10.1109/TCE.2009.5277999
9. Liangzhong Yin, Guohong Cao, Ying Cai: A generalized target-driven cache replacement policy for mobile environments. In: 2003 Symposium on Applications and the Internet, 2003. Proceedings. pp. 14–21 (Jan 2003). https://doi.org/10.1109/SAINT.2003.1183028
10. Perrin, M., Mullen, J., Helff, F., Gruenwald, L., d'Orazio, L.: Time-, energy-, and monetary cost-aware cache design for a mobile-cloud database system. In: Wang, F., Luo, G., Weng, C., Khan, A., Mitra, P., Yu, C. (eds.) Biomedical Data Management and Graph Online Querying. pp. 71–85. Springer International Publishing, Cham (2016)
11. Ren, Q., Dunham, M.H., Kumar, V.: Semantic caching and query processing. IEEE Transactions on Knowledge and Data Engineering **15**(1), 192–210 (Jan 2003). https://doi.org/10.1109/TKDE.2003.1161590
12. Sacco, G.M., Schkolnick, M.: Buffer management in relational database systems. ACM Trans. Database Syst. **11**(4), 473–498 (Dec 1986). https://doi.org/10.1145/7239.7336, http://doi.acm.org/10.1145/7239.7336
13. Stonebraker, M.: Operating system support for database management. Commun. ACM **24**(7), 412–418 (Jul 1981). https://doi.org/10.1145/358699.358703, http://doi.acm.org/10.1145/358699.358703
14. Xu, J., Hu, Q., Lee, D., Lee, W.C.: Saiu: An efficient cache replacement policy for wireless on-demand broadcasts. Proc. Ninth ACM Int'l Conf. Information and Knowledge Management (08 2000). https://doi.org/10.1145/354756.354785
15. Yin, L., Cao, G.: Supporting cooperative caching in ad hoc networks. In: IEEE INFOCOM 2004. vol. 4, pp. 2537–2547 vol.4 (March 2004). https://doi.org/10.1109/INFCOM.2004.1354674