Efficient workload offloading for mobile DBMS

Elif Gur Huseyin M Albayraktaroglu Hyungdae Yi
Department of Computer Science
Binghamton University
{egur2, halbayr1, hyi2}@binghamton.edu

ABSTRACT
Android smart phones have their own embedded DBMS(SQLITE). Although it can perform database operation on the smart phone without installing DBMS in the mobile, android smart phone experiences huge delay and energy consumption when it executes compute intensive DBMS operations. For compensate this problem, we developed an efficient database workload offloading algorithms which offloads compute intensive database operations to remote server DBMS. Our algorithm is deployed in an Android application called Research Information System. Our algorithms shows 222% ~ 1009% better performance than baseline approaches.

CCS Concepts
• Information systems → Retrieval on mobile devices • Information systems → Mobile information processing systems

Keywords
mobile database; workload offloading; mobile database architecture.

1. INTRODUCTION
Smart phone has high computation power and users can perform compute intensive task with smart phones. However, the compute intensive task requires much energy consumption of smart phone, so many research have conducted to offload the compute intensive tasks from mobile phones to remote servers. We discovered android embedded DBMS(SQLITE)'s performance is not enough to perform compute intensive SQL queries. For example, it take 170 second for an query in our system android phone, whereas the server can execute the same query in just 5 second. We developed system which can use both android DBMS and remote server DBMS. By the different nature of SQL queries, some queries are more efficient when it is executed in the mobile but some other queries are more efficient when it is executed in the server. Therefore, determining which side executes the query is important issues to be addressed. To handle this issue, we developed database offloading decision algorithms. We developed an android application adopting the offloading decision algorithms which is called Research Information System(RIS). We performed performance evaluation and the result show our approach has 222% ~ 1009% better performances against baseline approaches.

Section 2 describes our system architectures for the database offloading and RIS. Section 3 details our offloading decision algorithms. Section 4 shows performance evaluation results. We conclude this paper in Section 5.

2. SYSTEM ARCHITECTURES
We developed an android application to validate our database workload offloading algorithms. The application is Research Information System(RIS) which maintains research information database.

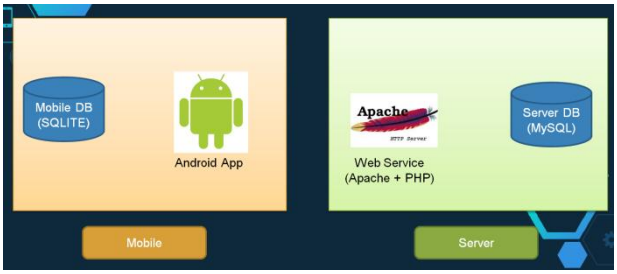

Figure 1. System Architecture of RIS.

2.1 System Overview
RIS can access embedded mobile DBMS and remote server DBMS which is connected via WiFi. The system architecture of RIS is shown in figure 1. Mobile side of RIS is composed of Android application and Android embedded mobile DBMS(SQLITE). Server side of RIS is composed of DBMS(MySQL) and web service to provide database connection between Android application and server DBMS.

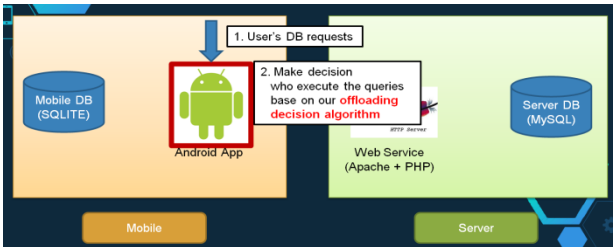

Figure 2-a. Offloading decision scenario 1.

Figure 2-b. Offloading decision scenario 2.

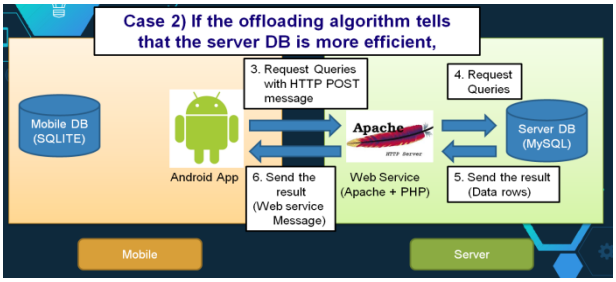

Figure 2-c. Offloading decision scenario 3.

RIS deployed offloading decision algorithms to determine which side's DBMS execute the queries. The detailed algorithms are described in the section 3. The offloading decision scenario is described in figure 2-a ~ 2-c. When the user requests DBMS queries, Android application decide which side(server or mobile) executes the query based on our database offloading decision algorithms. If the algorithm tells mobile side is more efficient, the android application request the query execution to the mobile DBMS and receive the result from the mobile DBMS. If the algorithm tells server side is more efficient, the android application request the query execution to the web service deployed in the server. The web service requests the query execution to the server DBMS and receives the result from the server DBMS and forwards the result to the android application. Web service is operated on Apache server and we developed PHP scripts for database connection and data transfer. Web service introduces overhead but android systems do not permit direct connection to mySQL for security reasons, so we deployed the web services.

2.2 Database Design

RIS maintains identical database on mobile and server. Figure 3 shows RIS database. RIS database has 5 tables which contains research information such as published papers and conferences. The record count of each table is different. The smallest table has 57 rows and the largest table has 221680 record count. By the different record count of each table facilitate developing different nature of queries for the application.

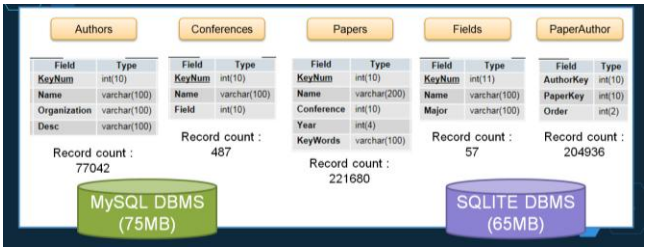

Figure 3. RIS database.

2.3 Queries

We developed 10 queries for the system to retrieve research information. RIS also can accept user's raw query. Some queries has simple database operation on small size table, but other queries has complex database operation such as aggregation and join on big size table. Figure 4 is a query example and Figure 5 is 10 queries we developed.

Query 1. Find all papers of an given author

```
String _Author_name;

SQL = "select A.name as author_name, B.name as Paper_name,
            B.conference as Conference, B.year as year,
            C.Order as author_order
      from authors A, Papers B, PaperAuthor C
      where A.keyNum = C.Authorkey
        and B.keyNum = C.Paperkey
        and A.name = '" + _Author_name +"'" ;
```

Figure 4. RIS query example(Query 1).

1. Find all papers of an given author.
2. Search all papers by an given keyword.
3. Find all fields of an major.
4. Find all papers which contains an phrase in its name.
5. Find conferences along with the number of papers published in a given year.
6. Find all majors along with the number of fields.
7. Find all author who published more than 10 papers.
8. Find all authors in an given organization.
9. Find the number of published papers of each year
10. Find all authors who published paper with an given keyword

Figure 4. 10 RIS queries.

3. OFFLOADING DECISION ALGORITMS

We developed offloading decision algorithm to make efficient decisions where to offload the Database workloads. Our algorithms base on profile collected in dynamic environment. We also developed re-profiling logic to compensate profiling errors.

3.1 Profiling Based Offloading Decision

We stores profiles in the android phone and the smart phone decide which tier(server or mobile) executes queries based on profiles. The profile indicates the expected execution time of each queries, so each query has its own profile. The profile is composed of Pm(Mobile side Profile) and Ps(Server side Profile). We denote Pm_{ni} as Profile of query n at ith execution in the mobile side and Ps_{ni} as Profile of query n at ith execution in the server side. We denote Em_{ni} as execution time of query n at ith execution in the mobile side and Es_{ni} as execution time of query n at ith execution in the server side. When a query is initially executed, our algorithm let the query is executed both in mobile DBMS and in server DBMS. The first profile is the query execution time of each side.

$$Pm_{n1} = Em_{n1} \qquad Ps_{n1} = Es_{n1}$$

After the first execution, the profile information is stored in the mobile phone. According to the profile, our offloading algorithm choose the faster side and executes the query only on the faster side. We update the profile after the execution using Exponentially Weighted Moving Average(EWMA) in order to reflect dynamic environment changes in wireless network. From the second execution of the query, the profile is updated using following formula.

$$Pm_{ni} = (1-a)Pm_{n(i-1)} + a \cdot Em_{n1}$$

$$Ps_{ni} = (1-a)Ps_{n(i-1)} + a \cdot Es_{n1}$$

The profile is fundamentally average of execution time of query. We assumes recent execution time of query reflect more precise environment, so we put more weight on the recent execution time using EWMA. We set a as 0.3.

3.2 Re-profiling

After the first profiling, our algorithm decide faster side according to profiled information, which means only one side will be executed all the time if profile of one side does not exceed the other side. The profile of the executed side reflect recent environment but the profile of the other side does not reflect recent environment. Therefore, the profile information might not indicate expected execution time. In order to address this challenge, we developed our algorithm to have a re-profiling procedure. If the consecutive execution of the one side exceeds

the threshold, the system performs re-profiling which execute the query both on the server and on the mobile and update the both profiles. The threshold is updated every execution of query using following formula.

$$\text{Threshold} = MAX(10,\ b \cdot \frac{|Ps_{ni} - Pm_{ni}|}{MIN(Ps_{ni},\ Pm_{ni})})$$

b is constant and we set b as 5 in our system. As b increases, threshold become larger. Re-profiling introduces overhead because it involves both-side query execution at the same query. In order to avoid large overhead, we set the minimum threshold as 10. Threshold increases when the difference of the server side profile and the mobile side profile is large.

4. PERFORMANCE EVALUATION

We deployed RIS in real environment. This section will show our performance evaluation. We validate our offloading decision algorithms in this section.

4.1 System Environment

We deployed RIS in a lab top server and mobile phones for performance evaluation as shown table 1. For server side system, we deployed RIS on Lenovo Thinkpad Laptop. We installed APM(Apache, PHP, MySQL) on the server. For mobile side system, we deployed RIS in 2 different phones to validate our dynamic offloading algorithms on different mobile phones. The first phone is an outdated android phone - Galaxy Nexus; the other phone is a cutting-edge android phone - Google Nexus 6P. Both phone have embedded DBMS(SQLITE), but Google Nexus 6P's SQLITE has better features because it has more recent android OS. We used Binghamton University's public WiFi as network between a server and the mobile phones.

Table 1. Performance Evaluation Hardware

Query	Server	Mobile 1	Mobile 2
Model	Lenovo Thinkpad X1	Galaxy Nexus	Google Nexus 6P
CPU	Intel core I7 2GHz 4 Cores	Cortex-A9 1.2GHz 2 Cores	Snapdragon 8.1 2GHz 8 Cores
RAM	8GByte	1Gbyte	3Gbyte
OS	Windows 7	Android 4.0	Android 6.0
Database	MySQL	SQLITE	SQLITE

4.2 Effectiveness of Database Offloading Decision Algorithms

Our offloading decision algorithm executes queries in the mobile side and in the server side at the first execution to get the initial profile. The first execution time of the server and of the mobile is the first profile(Pm_{n1},Ps_{n1}). We gathered the first profile under the environment described in the section 5.1, and the result is shown in the section 5.2. We observed big difference in mobile side execution time between outdated smart-phone(Galaxy Nexus) and cutting-edge smart-phone(Google Nexus 6P), but server side execution time is slightly faster in the cutting-edge smart phone. This result shows the mobile DBMS's performance is greatly affected by smart phone's hardware and android OS version.

Table 2. Offloading Decision in the First Execution

Query	Mobile 1(Galaxy Nexus)			Mobile 2(Google Nexus G6)		
	mobile	server	choice	mobile	server	choice
1	3ms	168ms	Mobile	19ms	121ms	Mobile
2	170458 ms	765 ms	Server	5950 ms	391 ms	Server
3	18 ms	165 ms	Mobile	6 ms	118 ms	Mobile
4	16251 ms	2471 ms	Server	853 ms	241 ms	Server
5	20643 ms	490 ms	Server	621 ms	304 ms	Server
6	5 ms	138 ms	Mobile	7 ms	73 ms	Mobile
7	84555 ms	630 ms	Server	2560 ms	223 ms	Server
8	9815 ms	199 ms	Server	520 ms	125 ms	Server
9	2707 ms	273 ms	Server	743 ms	213 ms	Server
10	96228 ms	5468 ms	Server	4643 ms	4940 ms	Mobile

The 10th Query's initial profile demonstrate why our offloading decision algorithm can make different offloading decision according to mobile phone's performance. In the same query, the outdated smart phone determines the server side execution is faster, but cutting edge smart phone determines the mobile side execution is faster. From the second execution of the query, each smart phone makes difference offloading decision on the same query; outdated smart phone - server, cutting-edge smart phone - mobile.

We compared our offloading decision algorithms to the baseline approaches. The first baseline approach is to execute all queries in the mobile side. We executed each queries 30 times and checked average execution time. Our offloading decision algorithm shows 230% ~ 1009% better average performance than the mobile only execution. For some queries, our approach suffer from worse performance because of profiling overhead which is resulted from the execution of both server-side and mobile-side. However, the performance gain from the offloading algorithm outweigh this overhead. The other baseline approach is to execute all queries in the server side. We also executed each queries 30 times and checked average execution time. Our offloading decision algorithm shows 222% ~ 230% better average performance than the mobile only execution.

Table 3. Offloading Decision Algorithm Vs Mobile-Side Only Execution

Query	Mobile 1(Galaxy Nexus)			Mobile 2(Google Nexus G6)		
	Mobile Only	Offloading algorithm	Comparison	Mobile Only	Offloading algorithm	Comparison
average			**1009%**			**230%**
1	3 ms	8 ms	-166%	19 ms	23 ms	-21%
2	170458 ms	6446 ms	2544%	5950 ms	589 ms	910%
3	18 ms	23ms	-27%	6 ms	9 ms	-50%
4	16251 ms	3012 ms	439%	853 ms	269 ms	217%
5	20643 ms	1178 ms	1600%	621 ms	324 ms	91%
6	5 ms	10 ms	-100%	7 ms	9 ms	-28%
7	84555 ms	3448 ms	2352%	2560 ms	308 ms	731%
8	9815 ms	520 ms	1787%	520 ms	142 ms	266%
9	2707 ms	360 ms	651%	743 ms	237 ms	213%
10	96228 ms	8675 ms	1009%	4643 ms	5904 ms	-27%

Table 4. Offloading Decision Algorithm Vs Server-Side Only Execution

Query	Mobile 1(Galaxy Nexus)			Mobile 2(Google Nexus G6)		
	Mobile Only	Offloading algorithm	Comparison	Mobile Only	Offloading algorithm	Comparison
average			**230%**			**222%**
1	168 ms	8 ms	2000%	121 ms	23 ms	426%
2	765 ms	6446 ms	- 742%	391 ms	589 ms	-33%
3	165 ms	23ms	617%	118 ms	9 ms	1210%
4	2471 ms	3012 ms	- 21 %	241 ms	269 ms	-11%
5	490 ms	1178 ms	- 140%	304 ms	324 ms	-6%
6	138 ms	10 ms	1280%	73 ms	9 ms	711%
7	630 ms	3448 ms	- 447%	223 ms	308 ms	-38%
8	199 ms	520 ms	- 161%	125 ms	142 ms	-13%
9	273 ms	360 ms	- 31 %	213 ms	237 ms	-11%
10	5468 ms	8675 ms	- 58%	4940 ms	5904 ms	-20%

4.3 Effectiveness of Re-profiling

Re-profiling introduces overhead because it requires both side execution at the same query. However, if the profiled execution time and the real execution time have big difference, the profiling based offloading decision algorithm without re-profiling might make worse decision. We executed query 9 in the cutting-edge smart-phone in the normal condition and the extreme condition as shown table 5. We made the extreme condition by putting heavy workload on the server such as video steaming, compiling and web browsing. If the server is under the extreme condition, the first profile stores 1570 ms as server side profile which is slower than mobile side execution. Assuming that mobile side execution time is pretty similar and the profiling is based on average execution time, the mobile side execution profile would never exceed the server profile if we don't adopt re-profiling. In normal condition, server side execution is better than mobile side execution. Therefore, the profiling based offloading decision approach makes always worse decision in this case. This example demonstrate why we need re-profiling even though it brings overhead.

Table 5. The first profile in difference condition of query 9

Condition	Mobile	Server	Choice
Normal	743ms	213ms	Server
Extreme	743ms	1570ms	Mobile

5. CONCLUSIONS

We developed database offloading decision algorithms which dynamically select which side execute the queries. Our algorithms are deployed in Research information system. Our offloading decision algorithms shows 222% ~ 1009% better performance than baseline approaches.