Pocket Data: The Need for TPC-MOBILE

Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek

University at Buffalo; Buffalo, NY 14260; USA
{okennedy,jerryant,challen,ziarek}@buffalo.edu
Website: http://odin.cse.buffalo.edu/research/

Abstract. Because embedded database engines, such as SQLite, provide a convenient data persistence layer, they have spread along with the applications using them to many types of systems, including interactive devices such as smartphones. Android, the most widely-distributed smartphone platform, both uses SQLite internally and provides interfaces encouraging apps to use SQLite to store their own private structured data. As a result, embedded database performance affects the response times and resource consumption of both the platforms that operation billions of smartphones and the millions of apps that run on them—making it more important than ever to characterize smartphone embedded database workloads. To do so, we present results from an experiment which recorded SQLite activity on 11 Android smartphones during one month of typical usage. Our analysis shows that Android SQLite usage produce queries and access patterns quite different from canonical server workloads. We argue that evaluating smartphone embedded database will require a new benchmarking suite, and we use our results to outline some of its characteristics.

Keywords: sqlite, client-side, android, smartphone, embedded database

1 Introduction

The world's 2 billion smartphones represent the most powerful and pervasive distributed system ever built. Open application marketplaces, such as the Google Play Store, have resulted in a vibrant software ecosystem comprising millions of smartphone and tablet apps in hundreds of different categories that both meet existing user needs and provide exciting novel capabilities. As mobile apps and devices become even more central to the personal computing experience, it is increasingly important to understand and improve their performance.

A common requirement of mobile apps and systems is persisting structured private data, a task that is frequently performed using an *embedded database* such as SQLite [18]. Android, the open-source and widely-used smartphone platform, provides interfaces that simplify the process of accessing private SQLite databases, and many apps make use of SQLite for this purpose. In addition, Android platform services themselves make heavy use of SQLite, as do built-in apps (Mail, Contacts), popular apps (Gmail, Maps), and libraries (Google Play Services) distributed by Google. As a result, the large and growing number of mobile apps using embedded databases represent a new and important class of database clients.

Unsurprisingly, mobile app usage of embedded databases is quite different from the workloads experienced by database servers supporting websites or big data applications. For example, while database servers are frequently tested and tuned for continuous high-throughput query processing, embedded databases experience lower-volume but bursty workloads produced by interactive use. As another example, enterprise database servers are frequently provisioned to have exclusive access to an entire machine, while apps using embedded databases compete for shared system resources with other apps and may be affected by system-wide policies that attempt to conserve limited energy on battery-constrained mobile devices. So while the fundamental challenges experienced by mobile apps using embedded databases—minimizing energy consumption, latency, and disk utilization—are familiar ground for database researchers, the specific tradeoffs produced by this domain's specific workload characteristics are far less well understood.

In this paper, we present results drawn from a one-month trace of SQLite activity on 11 PhoneLab [16] smartphones running the Android smartphone platform. Our analysis shows that the workloads experienced by SQLite on these phones differ substantially from the database workloads expressed by popular database benchmarking suites. We argue that a new benchmark for mobile embedded databases is required to effectively measure their performance, and that such a benchmark could spur innovation in this area.

Our specific contributions are as follows: (a) A month-long trace of SQLite usage under real world conditions (details in Section 2), (b) An in-depth analysis of the complexity (Section 3) and runtime (Section 4) characteristics of SQL statements evaluated by SQLite during this trace, (c) A comparison of these characteristics to existing benchmarking strategies (Section 5), and (d) An overview of the requirements for a new "pocket data" benchmark: TPC-MOBILE (Section 6).

2 Experimental Setup

To collect and analyze SQLite queries generated by Android, we utilized the unique capabilities of the PhoneLab smartphone platform testbed located at the University at Buffalo (UB). Approximately 200 UB students, faculty, and staff use instrumented LG Nexus 5 smartphones as their primary device and receive discounted service in return for providing data to smartphone experiments. PhoneLab participants are balanced between genders and distributed across ages, and thus representative of the broader smartphone user population. PhoneLab smartphones run a modified version of the Android Open Source Platform (AOSP) 4.4.4 "KitKat" including instrumentation and logging developed in collaboration with the mobile systems community. Participating smartphones log experimental results which are uploaded to a centralized server when the device is charging.

We instrumented the PhoneLab AOSP platform image to log SQLite activity by modifying the SQLite source code and distributing the updated binary library as an over-the-air (OTA) platform update to PhoneLab participants. Our logging recorded each SQL statement that was executed, along with its resulting runtime and the number of rows returned as appropriate. All current PhoneLab instrumentation including our SQLite logging statements are documented at `https://phone-lab.org/experiment/data/`. To protect participant privacy, our instrumentation removes as much personally-identifying information as possible, as well as recording prepared statement arguments only as hash values.

Our trace data-set is drawn from publicly-available data provided by 11 PhoneLab developers who willingly released[1] complete trace data for their phones for March 2015. Of the eleven participants, seven had phones that were participating in the SQLite experiment every day for the full month, with the remaining phones active for 1, 3, 14, and 19 days. A total of 254 phone/days of data were collected including 45,399,550 SQL statements. Of these, we were unable to interpret 308,752 statements (∼0.5%) due to a combination of data corruption and the use of unusual SQL syntax. Results presented in this paper that include SQL interpretation are based on the 45,090,798 queries that were successfully parsed.

[1] `https://phone-lab.org/static/experiment/sample_dataset.tgz`

3 Query Complexity

Operation	SELECT	INSERT	UPSERT	UPDATE	DELETE	Total
Count	33,470,310	1,953,279	7,376,648	1,041,967	1,248,594	45,090,798
Runtime (ms)	1.13	2.31	0.93	6.59	3.78	
Features Used						
OUTER JOIN	391,052				236	391,288
DISTINCT	1,888,013			25	5,586	1,893,624
LIMIT	1,165,096				422	1,165,518
ORDER BY	3,168,915				194	3,169,109
Aggregate	638,137			25	3,190	641,352
GROUP BY	438,919			25		438,944
UNION	13,801				65	13,866

Fig. 1: **Types and numbers of SQL statements executed during the trace, and query features used in each.**

In this section we discuss the query complexity we observed during our study and illustrate typical workloads over pocket data. Figure 1 summarizes all 45 million statements executed by SQLite over the 1 month period. As might be expected, `SELECT` forms almost three quarters of the workload by volume. `UPSERT` statements (*i.e.*, `INSERT OR REPLACE`) form a similarly substantial 16% of the workload — more than simple `INSERT` and `UPDATE` statements combined. Also of note is a surprising level of complexity in `DELETE` statements, many of which rely on nested sub-queries when determining which records to delete.

Client App	Statements Executed
Google Play services	14,813,949
Media Storage	13,592,982
Gmail	2,259,907
Google+	2,040,793
Facebook	1,272,779
Hangouts	974,349
Messenger	676,993
Calendar Storage	530,535
User Dictionary	252,650
Android System	237,154

(a)

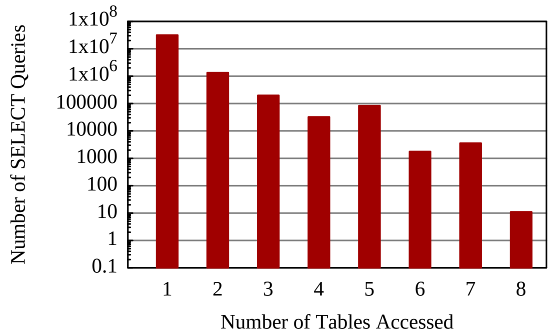
Client App	Statements Executed
Weather	12
Speedtest	11
KakaoStory	8
MX Player Pro	4
Quickoffice	4
VLC	4
Barcode Scanner	2
Office Mobile	2
PlayerPro	2
KBS kong	2

(b)

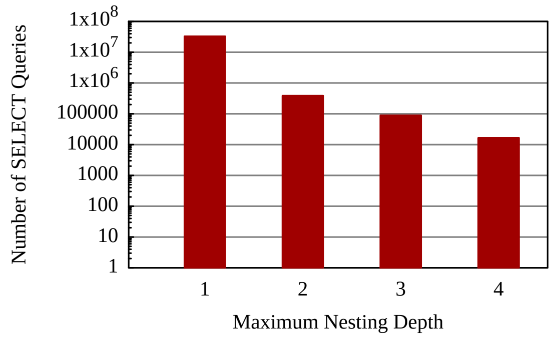
Fig. 2: **Apps that executed the (a) 10 most and (b) 10 fewest SQL statements.**

Figure 2 shows the 10 most frequent and 10 least frequent clients of SQLite over the one month trace. The most active SQLite clients include internal Android services that broker access to data shared between apps such as personal media, calendars and address books, as well as pre-installed and popular social media apps. There is less of a pattern at the low end, although several infrequent SQLite clients are themselves apps that may be used only infrequently, especially on a phone-sized device. We suspect that the distribution of apps would differ significantly for a tablet-sized device.

3.1 Database Reads



(a)



(b)

Fig. 3: `SELECT` **queries by (a) number of tables accessed and (b) maximum nesting depth.**

Of the 45 million queries analyzed, 33.47 million were read-only `SELECT` queries. Figure 3 shows the distribution of `SELECT` queries by number of tables accessed by the query, as well as the maximum level of query nesting. Nesting includes from-nesting (*e.g.*, `SELECT ... FROM (SELECT ...)`), as well as expression-nesting (*e.g.*, `SELECT ... WHERE EXISTS (SELECT ...)`). Even at this coarse-grained view of query complexity, the read-only portion of the embedded workload distinguishes itself from existing TPC benchmarks.

Like TPC-C [6], the vast majority of the workload involves simple, small requests for data that touch a small number of tables. 29.15 million, or about 87% of the `SELECT` queries were simple select-project-join queries. Of those, 28.72 million or about 86% of all queries were simple single-table scans or look-ups. In these queries, which form the bulk of SQLite's read workload, the query engine exists simply to provide an iterator over the relationally structured data it is being used to store. Conversely, the workload also has a tail that consists of complex, TPC-H-like [8] queries. Several hundred thousand queries involve at least 2 levels of nesting, and over a hundred thousand queries access 5 or more tables. As an extreme example, our trace includes 10 similar `SELECT` queries issued by the Google Play Games Service[2], each of which accesses up to 8 distinct tables to combine developer-provided game state, user preferences, device profile meta-data, and historical game-play results from the user.

Simple `SELECT` **Queries** We next examine more closely a class of *simple look-up* queries, defined as any `SELECT` query that consists exclusively of selections, projections, joins, limit, and order

[2] `https://developers.google.com/games/services/`

	Join Width					
Where Clauses	1	2	3	4	6	**Total**
0	1,085,154					**1,085,154**
1	26,932,632	9,105				**26,941,737**
2	1,806,843	279,811	5,970			**2,092,624**
3	384,406	80,183	29,101	1		**493,691**
4	115,107	70,891	10,696	939		**197,633**
5	28,347	15,061	1,162	17	11	**44,598**
6	212	524	591	471	3	**1,801**
7	349	22,574	333	1,048	8	**24,312**
8	35	18			6	**59**
9		541	2,564	4		**3,109**
10	159					**159**
11	545					**545**
Total	**30,353,789**	**478,708**	**50,417**	**2,480**	**28**	**30,885,422**

Fig. 4: **Number of simple look-up queries subdivided by join width (number of tables) and number of conjunctive terms in the `WHERE` clause.**

by clauses, and which does not contain any nested sub-queries or unions. Figure 4 shows queries of this class, broken down by the number of tables involved in the query (Join Width) and the complexity of the where clause, as measured in number of conjunctive terms (Where Clauses). For example, consider a query of the form: `SELECT R.A FROM R, S WHERE R.B = S.B AND S.C = 10` This query would have a join width of 2 (`R`, `S`) and 2 conjunctive terms (`R.B = S.B` and `S.C = 10`). For uniformity, `NATURAL JOIN` and `JOIN ON` (*e.g.*, `SELECT R.A from R JOIN S ON B`) expressions appearing in the `FROM` clause are rewritten into equivalent expressions in the `WHERE` clause.

The first column of this table indicates queries to a single relation. Just over 1 million queries were full table scans (0 where clauses), and just under 27 million queries involved only a single conjunctive term. This latter class constitutes the bulk of the simple query workload, at just over 87% of the simple look-up queries. Single-clause queries appear to be the norm. Recall that an N-way equi-join requires N-1 conjunctive terms; Spikes occur in the number of queries with one more term than strictly required to perform a join, suggesting a constraint on at least one relation.

Narrowing further, we examine simple look-up queries referencing only a single source table and a single conjunctive term in the WHERE clause. Figure 5 summarizes the structure of the predicate that appears in each of these queries. In this figure, constant terms (Const) are any primitive value term (*e.g.*, a quoted string, an integer, or a float), or any JDBC-style parameter (?). For simple relational comparators, we group together *in*equalities (*i.e.*, $<$, $\leq$, $>$, $\geq$ and $\neq$) under the symbol θ, and explicitly list equalities. Other relational operators such as `LIKE`, `BETWEEN`, and `IN` are also seen with some frequency. However, the majority (85% of all simple look-ups) are exact match look-ups. Not surprisingly, this suggests that the most common use-case for SQLite is as a relational key-value store. As we show shortly through a per-app analysis of the data (Section 3.1), 24 out of the 179 apps that we encountered posed no queries other than exact look-ups and full table scans.

Other `SELECT` Queries Figure 6 shows a similar breakdown for all 33.5 million `SELECT` queries seen. As before, the table shows the form of all expressions that appear as one of the conjunctive terms of a `WHERE` clause, alongside the number of queries where the expression appears. 31.0 million

Expression Type	Expression Form	Count
Exact Lookups	`Const = Expr`	26,303,579
Membership Test	`Expr [NOT] IN (List)`	331,788
Inequality on 1 constant	`Const` θ `Expr`	93,816
Patterned String Lookup	`Expr [NOT] LIKE Pattern`	72,289
Disjunction	`[NOT] Expr` $\vee$ `Expr`	61,541
Other Inequality	`Expr` θ `Expr`	38,714
Validity Test	`Expr IS [NOT] NULL`	17,305
No-op Clause	`Const or (Const = Const)`	6,710
Boolean Column Cast	`[NOT] Column`	5,358
Other Equality	`Expr = Expr`	1,471
Function Call	`Function(Expr)`	43
Range Test	`Expr BETWEEN Const AND Const`	18

Fig. 5: **The `WHERE` clause structure for single-tabled simple lookup queries with a single conjunctive term in the `WHERE` clause.**

of these queries contain an exact lookup. 1.6 million queries contain at least one multi-attribute equality expression such as an equi-join constraint, lining up nicely with the 1.7 million queries that reference at least two tables.

Expression Type	Expression Form	Count
Exact Lookups	`Const = Expr`	30,974,814
Other Equality	`Expr = Expr`	1,621,556
Membership Test	`Expr [NOT] IN (List or Query)`	1,041,611
Inequality on 1 constant	`Const` θ `Expr`	677,259
Disjunction	`[NOT] Expr` $\vee$ `Expr`	631,404
Bitwise AND	`Expr & Expr`	480,921
Other Inequality	`Expr` θ `Expr`	442,164
Boolean Column Cast	`[NOT] Column`	302,014
No-op Clause	`Const or (Const = Const)`	229,247
Patterned String Lookup	`Expr [NOT] LIKE Pattern`	156,309
Validity Test	`Expr IS [NOT] NULL`	87,873
Functional If-Then-Else	`CASE WHEN ...`	2,428
Range Test	`Expr BETWEEN Const AND Const`	2,393
Function Call	`Function(Expr)`	1,965
Subquery Membership	`[NOT] EXISTS (Query)`	1,584

Fig. 6: **WHERE clause expression structures, and the number of SELECT queries in which the structure appears as a conjunctive clause.**

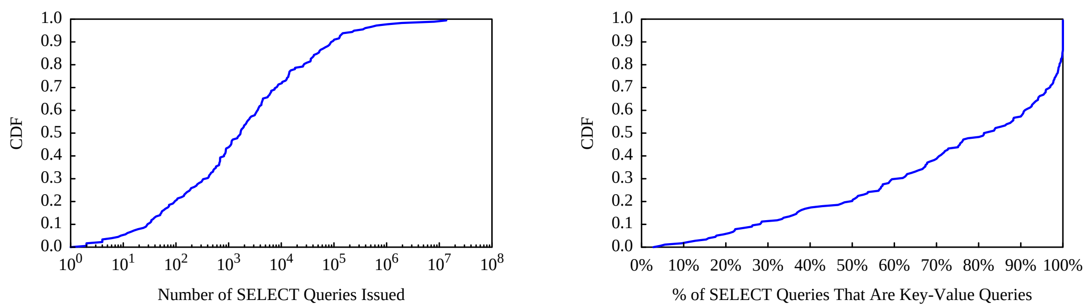
App developers make frequent use of SQLite's dynamic typing: Where clauses include bare column references (*e.g.*, `WHERE A`, implicitly equivalent to `WHERE A <> 0`) as well as bare bit-wise AND expressions (*e.g.*, `A&0xc4`). This latter predicate appearing in a half-million queries indicates extensive use of bit-arrays packed into integers.

Function	Call Sites	Function	Call Sites	Function	Call Sites
`GROUP_CONCAT`	583,474	`CAST`	38,208	`STRFTIME`	1,147
`SUM`	321,387	`UPPER`	20,487	`IFNULL`	657
`MAX`	314,970	`MIN`	19,566	`JULIANDAY`	587
`COUNT`	173,031	`COALESCE`	3,494	`DATE`	44
`LENGTH`	102,747	`LOWER`	3,110	`AVG`	15
`SUBSTR`	88,462	`PHONE_NUMBERS_EQUAL`	2,017		

Fig. 7: **Functions appearing in SELECT queries by number of times the function is used.**

Functions Functions extend the basic SQL syntax, providing for both specialized local data transformations, as well as computation of aggregate values. Figure 7 shows all functions appearing in `SELECT` queries during our trace, organized by the number of times that each function is used. All functions that we saw are either built-in SQLite functions, or in the case of `PHONE_NUMBERS_EQUAL` are Android-specific extensions; No user-defined functions appeared in the trace.

Overall, the most common class of function was aggregate functions (*e.g.*, `SUM`, `MAX`, `COUNT`), followed by string operations (*e.g.*, `LENGTH` and `SUBSTR`). The most commonly used function was `GROUP_CONCAT`, an aggregate operator that constructs a string by concatenating its input rows. This is significant, as it means that the most commonly used aggregate operator is holistic — its output size is linear in the number of input rows.



(a) (b)

Fig. 8: **Breakdown of `SELECT` queries by app. (a) Cumulative distribution of applications by the number of `SELECT` queries issued (note the logarithmic scale). (b) Cumulative distribution of applications by the percent of the app's `SELECT` queries that are full table scans or exact look-ups.**

Per-Application Analysis We next break the `SELECT` workload down by the calling application (app). Due to limitations of the logging infrastructure, 4.32 million queries (just over 12.9% of the workload) could not be associated with a specific application, and our app-specific analysis excludes these queries. Additionally, system services in Android are often implemented as independent apps and counted as such in the numbers presented.

Over the course of the one-month trace we observed 179 distinct apps, varying from built-in Android applications such as *Gmail* or *YouTube*, to video players such as *VLC*, to games such as

3 Kingdoms. Figure 8a shows the cumulative distribution of apps sorted by the number of queries that the app performs. The results are extremely skewed, with the top 10% of apps each posing more than 100 thousand queries over the one month trace. The most query-intensive system service, *Media Storage* was responsible for 13.57 million queries or just shy of 40 queries per minute per phone. The most query-intensive user-facing app was *Google+*, which performed 1.94 million queries over the course of the month or 5 queries per minute. At the other end of the spectrum, the bottom 10% of apps posed as few as 30 queries over the entire month.

We noted above that a large proportion of `SELECT` queries were exact look-ups, suggesting that many applications running on the device might be using SQLite as a simple key-value store. This suggestion was confirmed in our app-level analysis. For example, approximately half of one specific app's query workload consisted of the following two queries:

```
INSERT OR REPLACE INTO properties(property_key,property_value) VALUES (?,?);
SELECT property_value FROM properties WHERE property_key=?;
```

In this query, `?` is a prepared statement parameter that acts as a place holder for values that are bound when the prepared statement is evaluated.

To broaden the scope of our search for key/value queries, we define a key-value look-up query as a `SELECT` query over a single relation that either performs a full table scan, or performs an exact look-up on a single attribute. Figure 8b shows the cumulative distribution of apps sorted by the percent of its queries that are key-value lookup queries. For 24 apps (13.4%), we observed only key-value queries during the entire, month-long trace.

3.2 Database Writes

Write statements, `INSERT`, `INSERT OR REPLACE` (here abbreviated as `UPSERT`), `UPDATE`, and `DELETE`, together constitute 11.6 million statements or about 25% of the trace. As shown in Figure 1, the most prevalent operation is the `UPSERT`. `INSERT` and `UPSERT` together account for 9.3 million operations, of which 7.4 are `UPSERT`s. In many of these cases, the use of `UPSERTS` appears to be defensive programming on the part of wrapper libraries that make use of SQLite (*e.g.*, Object Relational Mappers, or ORMs). `UPSERTS` are also the canonical form of update in key-value stores, further supporting the argument that a large fragment of SQLite's traffic is based on key-value access patterns.

`DELETE` Statements The trace includes 1.25 million `DELETE` statements. This was by far the most expensive class of statement, with an average `DELETE` taking just under 4 ms to complete. A significant portion of this cost is attributable to the use of `DELETE` as a form of bulk erasure. As shown in Figure 9, 323 thousand `DELETE`s have no exact match condition in their WHERE clause, while 528 thousand do include a range predicate. `DELETE` predicates can become quite complex; 46,122 `DELETE`s (just under 3.7%) use nested `SELECT` queries, and touch as many as 7 separate tables (in 616 cases). This suggests extensive use of `DELETE` as a form of garbage-collection or cache invalidation, where the invalidation policy is expressed through SQL.

`UPDATE` Statements Slightly over 1 million statements executed by SQLite over the course of the month were `UPDATE` statements. Figure 10 breaks down the predicates used to select rows to be

Expression Type	**Expression Form**	**Count**
Exact Lookups	`Const` = `Expr`	926,042
Other Inequality	`Expr` θ `Expr`	527,517
Membership Test	`Expr [NOT] IN (List or Query)`	190,695
Disjunction	`[NOT] Expr` $\vee$ `Expr`	48,534
Inequality on 1 constant	`Const` θ `Expr`	31,128
Other Equality	`Expr` = `Expr`	10,037
Subquery Membership	`[NOT] EXISTS (Query)`	9,079
Boolean Column Cast	`[NOT] Column`	6,490
Patterned String Lookup	`Expr [NOT] LIKE Pattern`	6,109
Validity Test	`Expr IS [NOT] NULL`	2,693
Functional If-Then-Else	`CASE WHEN ...`	390
No-op Clause	`Const or (Const = Const)`	249
Range Test	`Expr BETWEEN Const AND Const`	18

Fig. 9: `WHERE` **clause expression structures, and the number of** `DELETE` **statements in which the structure appears.**

updated. Virtually all `UPDATE` statements involved an exact look-up. Of the million updates, 28 thousand did not include an exact look-up.

193 of the `UPDATE` statements relied on a nested `SELECT` statement as part of their `WHERE` clause, including 56 that involved 2 levels of nesting. Of the 193 `UPDATE`s with nested subqueries, 25 also involved aggregation.

Although the `WHERE` clause of the updates included a variety of expressions, *every single setter* in every `UPDATE` statement in the trace assigned a constant value; Not a single `UPDATE` expression attempted to compute new values using SQL, suggesting a strong preference for computing updated values in the application itself. This is not entirely unexpected, as the database lives in the address space of the application. Consequently, it is feasible to first perform a `SELECT` to read values out of the database and then perform an `UPDATE` to write out the changes, a tactic used by many ORMs. An unfortunate consequence of this tactic is that ORMs cache database objects at the application layer unnecessarily, suggesting that a stronger coupling between SQL and Java (*e.g.*, through language primitives like LINQ [2] or StatusQuo [4]) could be of significant benefit to Android developers.

Expression Type	**Expression Form**	**Count**
Exact Lookups	`Const` = `Expr`	1,013,697
Disjunction	`[NOT] Expr` $\vee$ `Expr`	84,937
Inequality on 1 constant	`Const` θ `Expr`	18,146
Membership Test	`Expr [NOT] IN (List or Query)`	14,146
Other Inequality	`Expr` θ `Expr`	9,443
Boolean Column Cast	`[NOT] Column`	1,640
Validity Test	`Expr IS [NOT] NULL`	1,517
Other Equality	`Expr` = `Expr`	221
Patterned String Lookup	`Expr [NOT] LIKE Pattern`	59

Fig. 10: `WHERE` **clause expression structures, and the number of** `UPDATE` **statements in which the structure appears.**

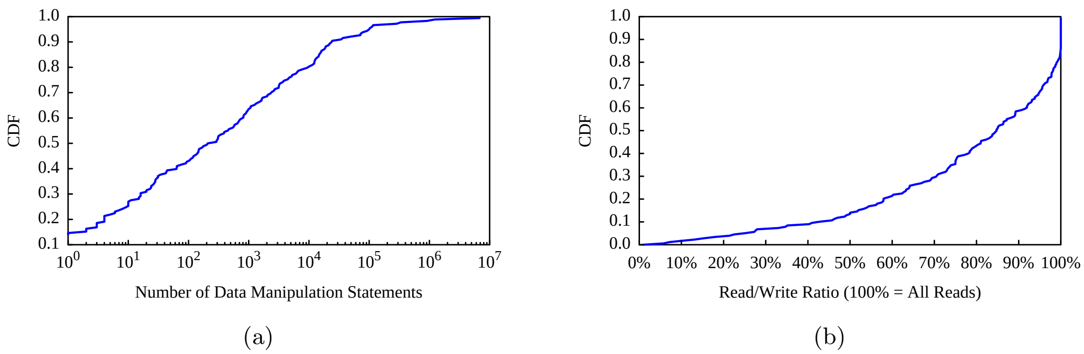


Fig. 11: **App-level write behavior. (a) Cumulative distribution of applications by number of data manipulation statements performed (note the logarithmic scale). (b) Cumulative distribution of applications by read/write ratio.**

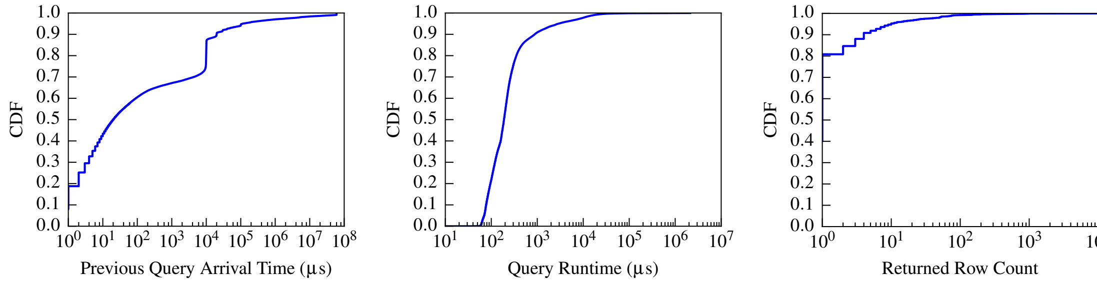
Per-Application Analysis Figure 11a illustrates app-level write workloads, sorting applications by the number of `INSERT`, `UPSERT`, `UPDATE`, and `DELETE` operations that could be attributed to each. The CDF is almost perfectly exponential, suggesting that the number of write statements performed by any given app follows a long-tailed distribution, a feature to be considered in the design of a pocket data benchmark.

Figure 11b breaks apps down by their read/write ratio. Surprisingly, 25 apps (14% of the apps seen) did not perform a single write over the course of the entire trace. Manual examination of these apps suggested two possible explanations. Several apps have reason to store state that is updated only infrequently. For example, *JuiceSSH* or *Key Chain* appear to use SQLite as a credential store. A second, far more interesting class of apps includes apps like *Google Play Newsstand*, *Eventbrite*, *Wifi Analyzer*, and *TuneIn Radio Pro*, which all have components that query data stored in the cloud. We suspect that the cloud data is being encapsulated into a pre-constructed SQLite database and being pushed to, or downloaded by the client applications. This type of behavior might be compared to a bulk ETL process or log shipment in a server-class database workload, except that here, the database has already been constructed. Pre-caching through database encapsulation is a unique feature of embedded databases, and one that is already being used in a substantial number of apps.

4 Runtime Characteristics

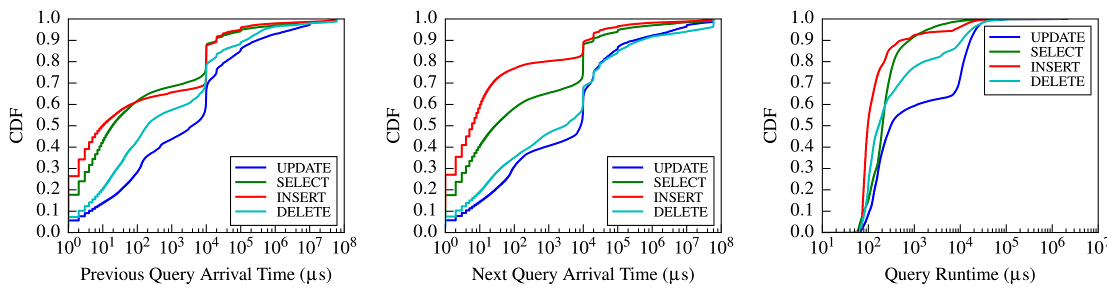
Next, we look at overall runtime characteristics of the query workload observed during our study. We examine how often queries arrive, how long they run, and how many rows they return—all important inputs into designing the TPC-Mobile embedded database benchmark.

General Characteristics Figure 12 shows query interarrival times, runtimes, and returned row counts (for `SELECT` statements) for all users, applications, and non-informational query types (`SELECT`, `UPDATE`, `INSERT`, `DELETE`) included in our dataset. Given that each mobile application is really generating an isolated workload to its own embedded database, we measure query interarrival time only between queries issued by the same application.



(a) (b) (c)

Fig. 12: **Summary Statistics for Android SQLite Queries. Distributions of (a) inter-query arrival times, (b) query runtimes, and (c) rows returned per query.**

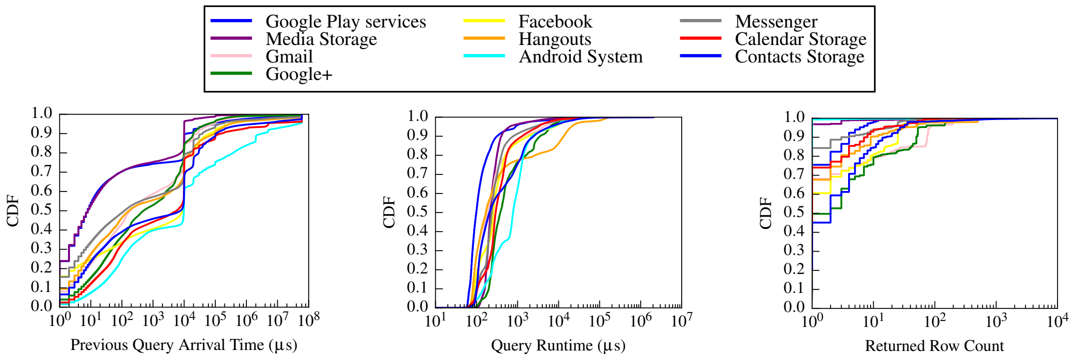


(a) (b) (c)

Fig. 13: **By-Query-Type Statistics for Android SQLite Queries. Distribution of times since the query (a) immediately preceding, and (b) immediately following the query in question. (c) Distribution of runtimes for each query.**

Examining the interarrival times shown in Figure 12a, it is interesting to observe that many queries seem to arrive much more quickly than the minimum query runtime shown in Figure 12b. Part of this may be due to apps that use multiple separate databases, which is not yet captured by our analysis. However, our logging is also done above any locking performed by SQLite, and so this may demonstrate that there are many cases where multiple application threads are issuing overlapping queries in parallel, even if the queries are eventually serialized before results are returned. Figure 12a also shows that, in addition to a standard long-tailed distribution of query inter-arrival times, about 20% of the workload is very periodic, arriving at a rate of 0.01 Hz.

The runtime CDF shown in Figure 12b shows while overall query runtimes show variation over several orders of magnitude, a large fraction of queries are executed in between 100 and 1000 μs. Further investigation into the small fraction of extremely slow queries may discover areas for database or application improvement. Finally, the row count CDF shown in Figure 12c shows that 80% of queries return only one row, further supporting our observation that many applications seem to be using the SQLite database almost as a key-value store.



(a) (b) (c)

Fig. 14: **Per-App Summary Statistics for Android SQLite Queries. Distributions of (a) inter-query arrival times, (b) query runtimes, and (c) rows returned per query.**

Runtime Characteristics by Query Type Figure 13 shows runtime characteristics for each of the four types of SQL statement. Figure 13a and 13b in particular show the time since the last query to be issued and the time until the next query is issued (respectively), while Figure 13c shows the distribution of runtimes for each type of query. Examining the differences between Figures 13a and 13b, we observe that `INSERT` queries are far more likely to arrive shortly before another query than shortly after. Almost 80% of `INSERT`s are followed by another query within 100μs. A similar, but far more subdued pattern can be seen for `UPDATE` statements. Conversely, both `SELECT` and `DELETE` statements are slightly more likely to arrive shortly before, rather than shortly after another query. Figure 13c shows significant deviations from the global average runtime for `DELETE` and `UPDATE` statements. `UPDATE` statements in particular have a bimodal distribution of runtimes, spiking at 100μs and 10ms. We suspect that this performance distribution is related to SQLite's use of filesystem primitives for locking and write-ahead logging [10, 11]. This could also help to explain the 0.01Hz query periodicity we observed above.

Runtime Characteristics by Application Figure 14 shows query interarrival times, runtimes, and returned row counts for ten of the most active SQLite clients. As seen in Figure 14a, the 0.01Hz periodicity is not unique to any one application, further suggesting filesystem locking as a culprit. Two of the most prolific SQLite clients, *Google Play services* and *Media Storage* appear to be very bursty: 70% of all statements for these applications are issued within 0.1ms of the previous statement. Also interesting is the curve for queries issued by the *Android System* itself. The interarrival time CDF appears to be almost precisely logarithmic for rates above 10μs, but has a notable lack of interarrival times in the 1ms to 10ms range. This could suggest caching effects, with the cache expiring after 1ms. As seen in Figure 14b, most apps hold to the average runtime of 100μs, with several notable exceptions. Over 50% of the *Android System*'s statements take on the order of 1ms. Just under 20% of *Hangouts* statements take 10ms, suggesting an update-heavy workload. Also, *Contacts Storage* has a heavier-duty workload, with 30% of statements taking between 100μs and 1ms. Figure 14c shows that the *Android System* and *Media Storage* issue almost exclusively single-row lookup queries. The remaining apps issue a large number of single-row queries — Even

Contacts Storage has a workload consisting of 45% single-row reads — the number of rows returned in general varies much more widely. Many of these apps' user interfaces have both a list and a search view that show multiple records at a time, suggesting that these views are backed directly by SQLite. Although all apps have long tails, two apps in particular: *Gmail* and *Google+* are notable for regularly issuing queries that return on the order of 100 rows.

5 Pocket Data and Related Work

In spite of the prevalence of SQL on mobile devices, and a increasing interest in so-called "small data" [9], relatively little attention has been paid to the rapidly growing *pocket data* space. In this section, we first explore some existing research on mobile databases, with a focus on how the authors evaluate their solutions. Then, we turn to existing benchmarking suites and identify specific disconnects that prevent them from being applied directly to model pocket data. In the process, we explore aspects of these benchmarks that could be drawn into a potential pocket data benchmark.

5.1 Pocket Data Management

Kang et. al. [11] explored the design of a flash-aware transactional layer called X-FTL, specifically targeting limitations of SQLite's redo logging on mobile devices. To evaluate their work, the authors used the TPC-C benchmark in conjunction with a series of micro-benchmarks that evaluate the file system's response to database write operations. This workload is appropriate for their target optimizations. However, as we discuss below, TPC-C is not sufficiently representative of a pocket data workload to be used as a general-purpose mobile database benchmark.

Jeong et. al. [10] noted similar limitations in SQLite's transactional layer, and went about streamlining the IO-stack, again primarily for the benefit of mobile devices. Again, micro-benchmarks played a significant role in the author's evaluation of their work. To evaluate their system's behavior under real-world conditions, the authors ran the *Twitter* and *Facebook* apps, simulating user behavior using a mobility trace generated by MobiGen [1]. This is perhaps the most representative benchmarking workload that we encountered in our survey of related work.

Many of the same issues with IO and power management that now appear in mobile phones have also historically arisen in sensor networks. Madden et. al.'s work on embedded databases with TinyDB [15] is emblematic of this space, where database solutions are driven by one or more specific target application domains. Naturally, evaluation benchmarks and metrics in sensor networks are typically derived from, and closely tied to the target domain.

5.2 Comparison to Existing Benchmarks

Given the plethora of available benchmarking software, it is reasonable to ask what a new benchmark for pocket-scale data management brings to the table. We next compare the assumptions and workload characteristics behind a variety of popular benchmarking suites against a potential TPC-MOBILE, and identify concerns that this benchmark would need to address in order to accurately capture the workload characteristics that we have observed.

Existing Mobile Benchmarks and Data Generators Although no explicit macro-benchmarks exist for mobile embedded databases, we note two benchmark data generators that do simulate several properties of interest: AndroBench [12] and MobiGen [1]. AndroBench is a micro-benchmark capable of simulating the IO behavior of SQLite under different workloads. It is primarily designed to evaluate the file-system supporting SQLite, rather than the embedded database itself. However, the structure of its micro-benchmark workloads can just as effectively be used to compare two embedded database implementations.

The second benchmark, MobiGen has little to do with data management directly. Rather, it generates realistic traces of environmental inputs, simulating the effects of a phone being carried through a physical space. Replaying these traces through a virtual machine running a realistic application workload could generate realistic conditions (*e.g.*, as in the evaluation of X-FTL [10]). However, it can not simulate the effects of user interactions with apps running on the device.

TPC-C One macro-benchmark suite that bears a close resemblance to the trace workload is TPC-C [6], which simulates a supply-chain management system. It includes a variety of transactional tasks ranging from low-latency user interactions for placing and querying orders, to longer-running batch processes that simulate order fulfillment. A key feature of this benchmark workload is the level of concurrency expected and required of the system. Much of the data is neatly partitioned, but the workload is designed to force a non-trivial level of cross-talk between partitions, making concurrency a bottleneck at higher throughputs. Conversely, mobile SQLite databases are isolated into specialized app-specific silos. In our experiments, throughput remained at very manageable levels from a concurrency standpoint. The most intensive database user, *Google Play services* had 14.8 million statements attributable to it, just under half of which were writes. This equates to about one write every 3 seconds, which is substantial from a power management and latency perspective, but not from the standpoint of concurrency.

YCSB We observed many applications using SQLite as a simple key/value store. Indeed, 13% of the applications we observed had a read workload that consisted exclusively of key/value queries, and over half of the applications we observed had a workload that consisted of at least 80% key/value queries. The Yahoo Cloud Services benchmark [5] is designed to capture a variety of key/value query workloads, and could provide a foundation for a pocket-scale data benchmark in this capacity. However, it would need to be extended with support for more complex queries over the same data.

Analytics These more complex queries include multiple levels of query nesting, wide joins, and extensive use of aggregation. As such, they more closely resemble analytics workload benchmarks such as TPC-H [8], The Star-Schema Benchmark [17], and TPC-DS [7]. This resemblance is more than passing; many of the more complex queries we encountered appeared to be preparing application runtime state for presentation to the user. For example the *Google Play Games* service tracks so-called *events* and *quests*, and participating *apps*. One of the most complex queries that we encountered appeared to be linking and summarizing these features together for presentation in a list view. We note that the presence of analytics queries in pocket data management is likely to increase further, as interest grows in smartphones as a platform for personal sensing [3, 13, 14].

TPC-E The TPC-E benchmark emulates a brokerage firm, and includes a mix of reporting and data mining queries alongside stream-monitoring queries. It models decision support systems that involve a high level of CPU and IO load, and that examine large volumes of rapidly changing data. SQLite does not presently target or support streaming or active database applications, although such functionality may become available as personal sensing becomes more prevalent.

6 Why TPC-MOBILE?

Our primary observation was that a pocket data workload includes a mix of both OLTP and OLAP characteristics. The majority of operations performed by SQLite were simple key-value manipulations and look-ups. However, a substantial fraction of the (comparatively read-heavy) workload consisted of far more complex OLAP-style operations involving wide, multi-table joins, nested sub-queries, complex selection predicates, and aggregation.

Many of these workload characteristics appeared to be motivated by factors unique to embedded databases. For example, SQLite uses single-file databases that have a standard, platform-independent format. As a consequence, we saw indications of entire databases, indexes and all, being transported in their entirety through web downloads or as attachments to other files [9]. A common pattern we observed was for a cloud service to package a fragment of its state into a SQLite database, which could then be cached locally on the device for lower-latency and offline access.

Query optimization goals also differ substantially for pocket data workloads. For example, latency is a primary concern, but at vastly different scales. Over our one-month trial, the average SQL statement took 2 ms to evaluate, and even complex `SELECT` queries with 4-level deep nesting only took an average of 120 ms.

Finally, unlike typical server-class benchmark workloads where throughput is a key factor, embedded databases have smaller workloads — on the order of hundreds of rows at most. Moreover, embedded databases need to share computing resources fairly with other processes on the same device. This means that in stark contrast to server-class workloads, an embedded database is idle more frequently. Periods of low-utilization are opportunities for background optimization, but must be managed against the needs of other applications running on the device, as well as the device's limited power budget.

Pocket data workloads represent a growing, and extremely important class of database consumers. Unfortunately, research and development on embedded databases (*e.g.*, [10, 11]) is presently obligated to rely on micro-benchmarks or anecdotal observations about the needs and requirements of embedded database engines. We believe that a new TPC-MOBILE benchmark that captures the characteristics observed in this paper can provide a principled, standardized way to evaluate advances in mobile database technology, which will in turn, help to drive the development of such advances.

7 Conclusions

In this paper, we identified embedded databases on smartphones as the foundation of a new class of *pocket data* workloads. We have presented the preliminary results for a long-running study of SQLite embedded database usage on Android smartphones, and identified numerous ways in which pocket data workloads differ from big data workloads. Through this study, we hope to be able to create a benchmark that will spur further research and development on pocket data and embedded databases.

References

1. Sabbir Ahmed. MobiGen: a mobility generator for environment aware mobility model. http://arrow.monash.edu.au/hdl/1959.1/109933, 2009.
2. Don Box and Anders Hejlsberg. LinQ: .NET language-integrated query. *MSDN Developer Centre*, 89, 2007.
3. A.T. Campbell, S.B. Eisenman, N.D. Lane, E. Miluzzo, R.A. Peterson, Hong Lu, Xiao Zheng, M. Musolesi, K. Fodor, and Gahng-Seop Ahn. The rise of people-centric sensing. *Internet Computing, IEEE*, 12(4):12–21, July 2008.
4. Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.
5. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, New York, NY, USA, 2010. ACM.
6. Transaction Processing Performance Council. TPC-C specification. http://www.tpc.org/tpcc/.
7. Transaction Processing Performance Council. TPC-DS specification. http://www.tpc.org/tpcds/.
8. Transaction Processing Performance Council. TPC-H specification. http://www.tpc.org/tpch/.
9. Jens Dittrich. The Case for Small Data Management. In *CIDR*, 2015.
10. Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *USENIX ATC*, pages 309–320, Berkeley, CA, USA, 2013. USENIX Association.
11. Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite databases. In *SIGMOD*, 2013.
12. Je-Min Kim and Jin-Soo Kim. AndroBench: Benchmarking the storage performance of Android-based mobile devices. In Sabo Sambath and Egui Zhu, editors, *Frontiers in Computer Education*, volume 133 of *Advances in Intelligent and Soft Computing*, pages 667–674. Springer Berlin Heidelberg, 2012.
13. Predrag Klasnja, Sunny Consolvo, David W McDonald, James A Landay, and Wanda Pratt. Using mobile & personal sensing technologies to support health behavior change in everyday life: lessons learned. In *AMIA*, 2009.
14. S.C.K. Lam, Kai Lap Wong, Kwok On Wong, Wenxiu Wong, and Wai Ho Mow. A smartphone-centric platform for personal health monitoring using wireless wearable biosensors. In *ICICS*, Dec 2009.
15. Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM TODS*, 30(1):122–173, March 2005.
16. Anandatirtha Nandugudi, Anudipa Maiti, Taeyeon Ki, Fatih Bulut, Murat Demirbas, Tevfik Kosar, Chunming Qiao, Steven Y. Ko, and Geoffrey Challen. PhoneLab: A large programmable smartphone testbed. In *SenseMine*, pages 4:1–4:6, 2013.
17. Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin Heidelberg, 2009.
18. Mike Owens and Grant Allen. *SQLite*. Springer, 2010.