Mobile Netw Appl (2013) 18:42–59
DOI 10.1007/s11036-012-0413-z

A Mobile App Search Engine

Anindya Datta · Sangaralingam Kajanan · Nargis Pervin

Published online: 11 October 2012

Abstract With the popularity of mobile apps on mobile devices based on iOS, Android, Blackberry and Windows Phone operating systems, the numbers of mobile apps in each of the respective native app stores are increasing in leaps and bounds. Currently there are close to one million mobile apps across these four major native app stores. Due to the enormous number of apps, both the constituents in the app ecosytem, consumers and app developers, face problems in 'app discovery'. For consumers, it is a daunting task to discover the apps they like and need among the huge number of available apps. Likewise, for developers, enabling their apps to be discovered is a challenge. To address these issues, Mobilewalla (MW) an app search engine provides an independent unbiased search for mobile apps with semantic search capabilities. It has also developed an objective scoring mechanism based on user and developer involvement with an app. The scoring mechanism enables MW to provide a number of other ways to discover apps—such as dynamically maintained 'hot' lists and 'fast rising' lists. In this paper, we describe the challenges of developing the MW platform and how these challenges have been mitigated. Lastly, we demonstrate some of the key functionalities of MW.

Keywords Mobile app · Search engine · Semantic similarity · App discovery · Scoring

A. Datta (✉) · S. Kajanan · N. Pervin
School of Computing, National University of Singapore,
13 Computing Drive, 117417 Singapore, Singapore
e-mail: datta@comp.nus.edu.sg

S. Kajanan
e-mail: skajanan@comp.nus.edu.sg

N. Pervin
e-mail: nargisp@comp.nus.edu.sg

1 Introduction

Consumer software applications that run on smartphones (popularly known as mobile apps, or, simply, apps) represent the fastest growing consumer product segment in the annals of human merchandising [1, 2]. The absolute number of apps currently in existence, as well as their rates of growth, are remarkable. At the time of writing this paper, there are 586,822, 316,403, 43,277, 35,171 number of apps available in Apple, Android, Blackberry and Windows platforms respectively. Since December, 2010, each month the app growth rates for the Apple and Android platforms has been nearly 4 % and 7 % respectively.

The huge number of apps and their increasing growth rate has created number of problems for the key constituents of app ecosystem. For consumers, there are simply too many apps and far too much fragmentation in these apps (e.g., a large number of categories). The analogy we often use to describe the confusion faced by a mobile app consumer is to imagine a customer walking into a grocery store, needing only a few items, and finding that all aisles and category labels have been eliminated, and every product has been thrown into a pile on the floor. In the same way

it is a daunting task for a consumer to navigate through the native app stores [3, 4] and discover apps they need and like. This issue has raised concerns in the recent media forums [5, 6].

The situation of the app developer is even worse. There are over 900,000 mobile apps in Apple and Android alone and most smartphone owners can only identify a handful—this is a nightmare scenario for developers whose success is contingent upon getting their apps "found" by consumers. "How will my apps be discovered?" is the number one question in the minds of app developers. This issue, known as the "app discovery" problem, has received wide attention in the media as well [7, 8].

Thus, it highlights the need to have an effective system for search and discover the applications—in essence, a "search engine" for apps. The reader might point out that each of the native app markets (e.g., the iTunes appstore and the Android market) offer search capabilities. However, as has been widely discussed [9, 10], the native app stores are commercially driven and search results are highly influenced by what the store wants the consumer to see, rather than being unbiased on producing relevant output. Moreover, as has also been widely reported, app store designs are confusing along many dimensions, such as having multiple display lists where individual apps are often listed in conflicting orders. For example, at the time of writing this paper 'mSpot Music' was the first featured app in the Android market [3]. This app is in the category 'Music & Audio'. Investigating further in the 'Music & Audio' category of the Android market, we found that 'mSpot Music' is not even in the top 200 apps in that category.

Therefore, there is intense need of creating unbiased search systems for mobile apps. One of the earliest entrants in this space is Mobilewalla (MW) (www.mobilewalla.com). MW is a full fledged app search engine employing semantic search capabilities. In addition to supporting basic keyword search, MW also features a number of other ways to discover apps—such as dynamically maintained "hot" lists and "fast rising" lists. One of MW's major contributions is the creation of an unbiased scoring system for apps based on the Mobilewalla Score (MWS). MWS has been recently featured in a number of "top app" lists, the most prominent being the New York Times Box Scores [11].

In this paper we describe Mobilewalla platform. In Section 3 and 4, we describe the architecture and approach. In Section 5, we explain the functionality of Mobilewalla platform with some screen shots. In Section 6, we discuss some of the other app search platforms and compare those with Mobilewalla system. Lastly in Section 7, we conclude the paper.

2 Overview

As articulated previously, Mobilewalla has been developed as a search, discovery and analytics system for mobile apps. Mobilewalla consists of three independent components.

1. A data collection (DC) component, which collects data from native app stores (Apple iTunes [4], Android Market [3], Blackberry App World [12] and Windows Mobile Marketplace [13]).
2. The Computational Backend (CB), which cleans and processes data collected by the DC.
3. An user interface and query handler (UI-QH), which displays data that has been collected in step 1 and computed in step 2, and also handles user queries from the user interface.

All of these components were created to address a number of challenges, which we outline below.

1. Developing automated extraction mechanisms, or *crawlers* that work on dynamic web sites.
2. Uniform categorizations across different stores, handling mischaracterization of app content by developers and automated *data cleaning*.
3. Allowing real-time search of vast app content
4. Computing "similarity" between, and across, mobile apps
5. Effectively presenting a vast amount of multi-dimensional information to the user, which in turn would result in effective searching of apps and the discovery of apps relevant to the user's requirements.

Below, we describe the high level architecture of the complete Mobilewalla system, and elaborate on the solution methodologies employed to overcome the above challenges.

3 Architecture

Figure 1 shows the architecture of the Mobilewalla platform. The Data Collector (DC) component gathers information from the native app stores, extracts data and stores it in the Mobilewalla database (MWDB). The Computational Backend (CB) interacts with the MWDB to perform computations which enable users to search and discover apps efficiently. Finally, the UI-QH component interacts with the application server to retrieve the data from MWDB and present it to users.

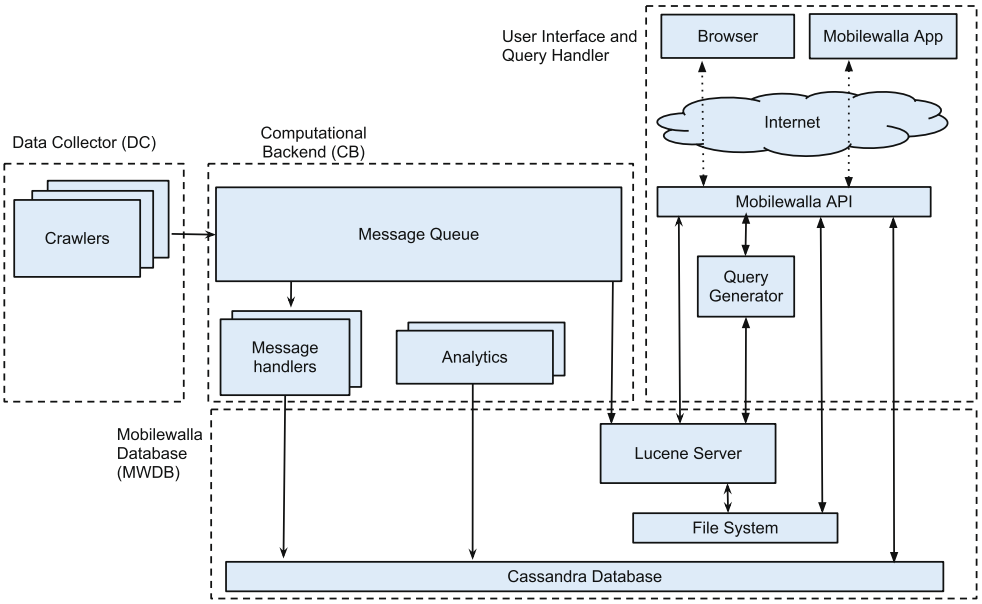

Fig. 1 Logical architecture of Mobilewalla

3.1 Data collector (DC)

The essence of the DC component is an automated extraction system, or a *crawler*, that fetches useful data out of the native app stores and writes it to the MWDB. We first describe the crawler and then discuss the MWDB.

Designing crawlers for app stores are not an easy task, as the app stores are not based on a uniform structure of data representation. Rather, each app store has its own format. Therefore, we have developed four different crawlers, one for each of the four native app stores—(i) iStore [4], (ii) Android Market [3], (iii) Blackberry App World [12] and (iv) Windows Mobile Marketplace [13]. Typical crawlers [14–16], that are widely available as open source software components, have been developed to fetch information from generic internet sites. These crawlers work on the assumption that web content is available in static HTML pages and do not work on dynamically generated content created with technology such as AJAX [17], Silverlight [18] and Flash ActionScript [19]. The native apps stores are built using these dynamic web technologies. These technologies are based on scripting languages, which create component based web pages on the fly. Let us consider the example of AJAX, arguably the most prevalent dynamic web development framework. AJAX is dependent on a scripting language called Javascript, which allows dynamic update of parts of a web page. This makes it difficult to download and parse the complete web page as is usually done in standard crawlers. Consider the example of the Google search engine. Google can crawl static content such as that can be found in HTML pages, PDF and Word documents. Google is unable to crawl an AJAX based web site such as Kayak.com [20]. For a AJAX based web application, the response from the web server comes in the form of JSON [21] and not in the form of HTML. To crawl and extract information from a AJAX based web site, one needs to parse JSON responses.

Complicating matters further, each app store has its own JSON response format. To handle this, we have developed ways to parse these individual JSON responses to extract app specific details based on the JQuery library [22]. Consequently, we have had to develop individual store specific crawlers that encapsulate the knowledge of the JSON format for each individual store. Understanding the JSON format for each store and developing "personalized" parser routines has proven to be a non-trivial task.

Having described our idea behind our automated extraction mechanism, we now focus on the specific data item that we extract. These are shown in Table 1.

Our extracted information consists of both static and dynamic data elements as shown in Table 1. This data is extracted and stored in the Mobilewalla database (MWDB) as described below in Section 3.2.

The static data is downloaded and stored in the database only once. If there occurs any changes to this data the updated information is downloaded again and the old data is replaced with this new information. For example, an app might have its description updated. In this case, the old description is overwritten in the database with the new description.

Table 1 Data captured by crawler

Static data	App name
	App description
	Release date
	Platform
	Price
	Screen shot
	Icon
	Category
	Seller/developer name
	URL
	Language
	Size of an app (MB)
Dynamic data	Comment
	Rating
	Version
	Rank in the store
	Times downloaded

Dynamic data, which typically changes frequently, is fetched by the crawler on a continuous basis. For dynamic information the old data is never deleted or replaced. Rather, all dynamic data is time stamped and retained in the database. As an example, consider the user rating of an application, an important dynamic data item. Whenever there is a change in an application's ratings in its native app store, the new rating is stored along with a date stamp. Similarly, any new comment, if discovered, are appended to the database in the same manner.

When a new app is encountered, the crawler downloads all data related to this app. The crawler then runs regularly to track data about this app. How frequently the crawler will revisit a particular app is determined dynamically. Let us assume, T_a denotes the interval at which the crawler will revisit a particular app a. Initially $T_a = 3$ h. If in the next iteration the crawler does not find any update for the app a, the next revisit will happen at $n \times T_a$ period, i.e. T_a will be updated as $T_a^{\text{new}} = n \times T_a^{\text{old}}$, where n is a positive number. If even in the subsequent visit, the crawler does not find any new update or new dynamic data, then T_a will be updated again. At maximum value of $T_a = T_a^{\max}$, the value of T_a is no more updated. We set $T_a^{\max} =$ 48 h, i.e. at every 2 days the crawler will revisit each app irrespective of whether there is any update or not for that particular app. Such an approach ensures two aspects of the crawler. (i) The crawler need not visit all the apps in a store at every iteration, which reduces the total time the crawler takes at every run. In our environment, this enables the crawler to run at every 3 h. For highly dynamic and popular apps, this in turn guarantees that the crawler will quickly (within 3 h) identify the new data and put into our system. (ii) For less dynamic apps, the crawler will identify the new data within a maximum 2 days. This system creates a balance between dynamism of the app and the scalability of the crawler system.

In many instances, existing apps get deleted from its native stores. In these circumstances, the crawler will fail to download any data from the store. We keep a note of how many consecutive times the crawler is failing to download the information related to a particular app. If the count crosses a threshold, we assume that the app has been deleted from the store and mark it as inactive in our database.

The physical architecture of MW system is shown in Fig. 2. The crawlers are deployed in 4 small machines distributed geographically across the world in USA, Canada, Europe and India. Each of these machines has a configuration of 2 GB RAM, 40 GB Hard disk and 1 i7 quad-core processor.

3.2 The Mobilewalla database

Having described the crawler system, we now delve into the Mobilewalla database (MWDB), which is the store-house of the crawler extracted information. The MWDB has three components—a distributed replicated no-sql database based on Cassandra [23], file system and an unstructured data store based on Lucene text search engine [24].

No-SQL database MW crawler collects data from native app stores. The native app stores are country specific. So the Apple store for India is different than the Apple store in USA. This necessitates crawling different country specific native app stores and storing this data. Due to this large number of stores per native app store, the approximate data size per week collected by MW crawler is in the range of 1 TB. Over the last one year, the data size of Mobilewalla across all countries has grown over 50 TB. It is difficult to store and manage such a growing data in a traditional relational database. So, following the current trend in big data [25], we resort to one of the big data no-sql database, the Cassandra [23]. As depicted in Fig. 2, the Cassandra is deployed in the two physical machines as the two replica groups.

The Cassandra database of MWDB primarily contains structured data captured by the DC such as Version Number, Release Date, Price and Ratings. The Cassandra database is structured in column family. Instead of describing each column family, we present the `app` column family only in Table 2. The rows in column family `app` reflects the information captured in Table 1. As can be seen in the `app` column family, the ratings,

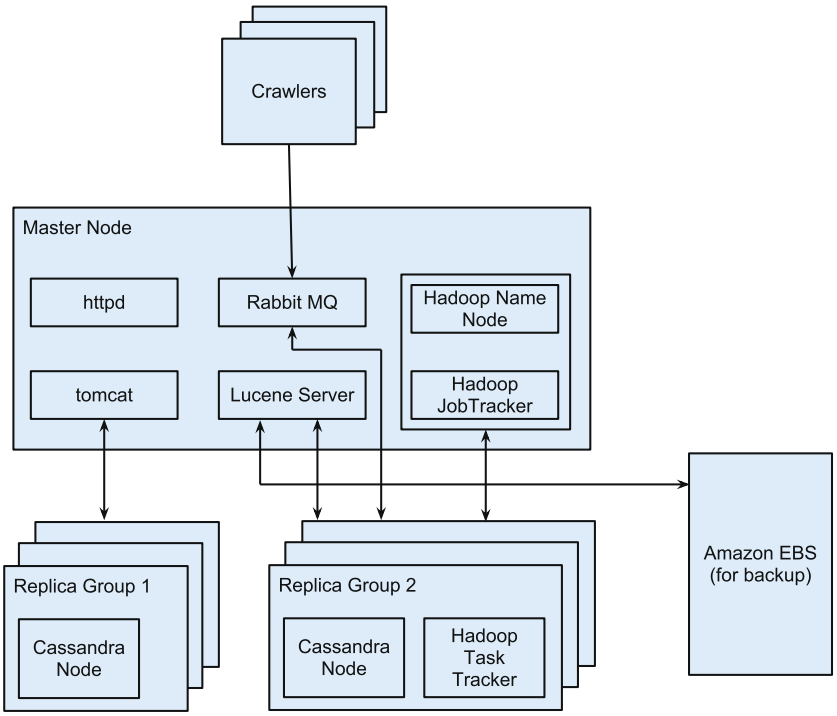

Fig. 2 Physical deployment architecture of Mobilewalla

the ranks and the prices are stored as country specific rows in the `app` column family.

In the `app` column family, in addition to the store specific category `r-primary-category`, one should note the row `mwcategories`. One of the challenges encountered in Mobilewalla was how to enable users to browse apps by categories such as Finance, Entertainment or Life Style. Native app stores designate categories for each app; however these store specified categories have a number of issues. (1) Developers tend to wrongly categorize apps based on where it would attract most users rather than the category to which the app naturally belongs based on its content. This often results in gross miscategorization of apps. For instance, the android app 'Men and women of passion video' is in the category 'Health & Fitness' in the Android market, whereas this app is an adult video app, and should have been appropriately categorized under 'Media & Video'. Natives stores themselves often do not perform extensive verification of such errors. One of our goals in Mobilewalla was to remove these miscategorizations as far as possible. (2) Another major issue is the lack of uniform categorization across the stores. For instance, 'Skype' is under 'Communication' category in Android market, whereas it is under 'Social Networking' category in iStore. Our goal in Mobilewalla was to present apps based on a uniform ontology.

To address this issue, we have developed an unique categorization scheme across multiple stores, called mw categories. The information about these mw categories are kept in the `mw_category` column family (not shown Table 2). We have developed an automatic categorization scheme, which is based on the Wikipedia ontology [26]. At a high level, we use the title and the description of an app to identify the keywords for an app. These keywords are matched with the keywords for Wikipedia categories to categorize the app in a mw category defined in `mw_category` column family.

Lucene A key requirement in Mobilewalla was to support free form keyword search. We use the Lucene text search engine to fulfill this requirement. In particular we create an inverted index out of the textual content attached to each app (including the description and the title). Subsequently we support search based on this index.

We should point out however that a straight forward implementation on top of Lucene was not possible. The reasons are,

1. The Lucene query processor awards equal weights to every keywords in a document. This does not work for us, as we have developed certain proprietary rules about the importance of different keywords. (e.g., keywords in the title should have higher weights than keywords in the description) To handle this we modified Lucene such that variable weights could be attached to keywords.

Table 2 Column family "app" in Cassandra

```
column family: app
row key: <appid>
columns:
name: <value>
description: <value>
version: <value>
mwscore-<country code>: <value>
mwcategories: [<mw_category>, <mw_category>...]
last-update-<country code>: <value>
price-<country code>: <value>
developer: <value>
developerID: <value>
seller: <value>
update-date: <value>
size: <value>
languages: <value> [string]
url-<country code>: <value>
rank-<country code>-<category name>: <value>
rating-cv-all-count-<country code>: <value>
rating-cv-all-star-<country code>: <value>
rating-av-all-count-<country code>: <value>
rating-av-all-star-<country code>: <value>
release-date: <value>
release-date-flag: <crawler|store>
r-countries-<country code>: null
r-categories-<local category code>: null
r-primary-category: <local category code>
r-device: <device>
download-range: <value>
version-required: <value>
permissions: <value>
whats-new: <value>
video-url: <value>
```

2. Lucene does not perform stemming, i.e. if a user searches for the word "swimming" the apps related to the keyword "swim" will not be returned by Lucene. We incorporated stemming and lemmatization to transform a word in to its base form. The stemming uses simple word transformation to modify a word to its base form, e.g. "going" to "go". The lemmatizer uses the word's parts of speech (POS) and a valid dictionary to identify the original form. Our lemmatizer is based on the OpenNLP [27] parser and the Wordnet dictionary. We used both the original word and the transformed base form of the word to index the app in the Lucene database. As depicted in Fig. 2, Lucene is physically deployed in the master node, where we run all the Mobilewalla computational tasks.

File system The MW crawler collects image icons and screen shots related to each app from native app stores. Due to the size of icons and screen shots, instead of storing them in the database, we store these images into a file system. As shown in Fig. 2, the MW file system is based on Amazon's Elastic Block Store (EBS) [28]. The file system also works as the persistent storage for Lucene.

3.3 Computational Backend (CB)

The Computational Backend (CB) receives the base data gathered by crawler in the form of a message queue. The message handler processes the messages in the message queue, cleans the data and stores it in the Cassandra database. The message queue architecture allows us to run the crawler and persist the data in the Cassandra database asynchronously. Another important component of the CB is the analytics, which produces a set of *proprietary metrics*.

Data cleaning The raw data captured by the crawler has been primarily entered by the developers who typically make a number of (often intentional) errors. The data cleaning component of the CB attempts to fix these errors as follows.

Categorization—We discussed the issues related to store categories before. As discussed, we can't rely on the store and developer provided categories. So we needed to develop our own automatic categorization system. A key task of the CB is to perform the Wikipedia based categorization described before.

Duplicate App Removal—Often app developers submit an app in a store and for the next version the app developers do not update the already existing app, rather they create a new entry in the system. This creates duplicate entry of the same app in the store. In iPhone stores we found that about 10,000 such apps exist. We have developed an automatic identification method of the duplicate apps based on comparing core app attributes such as app title, description and developer name.

Deleted or Inactive App—Many apps in native stores turn out to be orphaned or defunct. These apps, after release, have not had any version upgrades or platform upgrades, likely because the developer abandoned the app. We have developed a heuristic based solution to identify these defunct apps based on inactivity periods and platform compatabilities.

Mobilewalla Score (MWS) One of the values Mobilewalla provides to the end user is providing a unique and uniform scoring mechanism of apps across categories and platforms. Unlike existing app search engines, where app rankings are based on highly subjective

and often commercial considerations, in Mobilewalla we wanted to rate and rank apps based on uniform unbiased criterion. This is achieved through the Mobilewalla Score (MWS). The MWS of an app is based on several objective parameters that denote how users and developers are engaged with the app. Some of the factors used to compute the MWS are (1) the current and historical ratings of the app in its native store, (2) the frequency of releases, (3) the duration that the app is active in the store since its first release, (4) number of users who rated the app, (5) number of users who provided comments on this app, (6) the current and historical rank of the app in its native store, (7) the number of apps in the category the app belongs to, and (8) the past history of the developer in producing apps. The system computes the MWS every night and posts it in the database. We keep track of the historical value of MWSs for each app.

The current and historical value of MWS are used to create a number of popular lists of Mobilewalla.

Hot Apps—This is the list of apps with the highest MWS at current time.

Fast Movers—This is the list of the apps for which the change of the MWS is the highest. This measures the acceleration in the MWS for each app and report top apps with the acceleration rate.

All Time Greats—This is the list of the apps which have the highest average MWS over last 3 months. The MWS is also used to compute a Mobilewalla developer score (MWDS), which may be used to denote the overall success of a developer in producing mobile apps. The detailed computation of the MWS and MWDS is given in Section 4.

Similarity computation Mobilewalla enables users to find apps similar to a particular app and compare the similar apps on characteristics such as MWS and price (much like the "if you like this you might also like" feature in Amazon). We compute a measure of similarity between two apps based on the semantic similarity across several parameters, such as description, title and comments received from users. The semantic similarity computation of these features are done using hypernym, hyponym and synonym relationships in Wordnet [29]. Based on the importance in describing features of an app, the similarity measurement across each of these parameters is given different weight in the final similarity measurement. For example, the similarity on title is given more weight, than the similarity on description. The similarity on comment is given the least weight. For each app, we identify the similar apps that cross a threshold on similarity measurement with respect to that app. The exact details of the similarity computation algorithm is proprietary to Mobilewalla and so is not described in detail here due to IP issues.

In-memory index and index builder One of the objectives of the Mobilewalla application is to enable complex search in the database in real time. For this, the in-memory index contains materialized views of the database containing following items (1) The inverted index of the Lucene data and (2) Index of the app on the basis of app parameters, such as price, release date, and developer name. These indexes are pre-built nightly, and reduce the complex join operations that require querying the database against some of the user queries.

The message queue and message handler are implemented using RabbitMQ [30]. The analytics is implemented using Hadoop Job Tracker [31] and Hadoop NameNode system [32]. The RabbitMQ, Hadoop JobTracker and Hadoop NameNode, all of them run in the same master node along with the web server (httpd server) and the application server (apache tomcat) (Fig. 2). Most of the analytics such as similarity computation, computation of hot app, computaton of fast movers app and, computation of Mobilewalla Score (MWS) are done off-line in batch mode. The Hadoop Job Tracker and Name nodes are responsible for distributing the computation across multiple nodes in replica group 2 by MapReduce framework. The nodes in replica group 2 do the actual computation and persist the some of the computed output (such as MWS) in the cassandra database in the node. Other computational output such as specialized list of apps (e.g., hot apps) are fed back to the Hadoop Job Tracker in the master node to consolidate (reduce of MapRedue) and store it in memory for fast access by users. Because the master node does not do much of the computation the overhead due to Hadoop JobTracker and NameNode is very minimal in the master node.

3.4 User interface and query handler

Users can interact with the MW system using a browser (desktop or mobile device based browser) or mobile apps in Android and iOS platform. Both the MW app and web site interact with the MW system over the internet, using JSON format. The JSON request and response is handled by the apache tomcat application server. Most of the incoming requests from the users through MW mobile app or web site can be fulfilled by directly fetching data from MWDB. The keyword based search is handled through query generator.

The query generator module receives user provided query keywords for searching the app database. The query generator transforms this user query across several dimensions to generate more than one query. The result of the query generator is a series of queries ordered in priority. These queries are executed in Lucene's in-memory index to identify the app. If the prior queries result in a threshold number of relevant apps from the Lucene in-memory index, the later queries are not executed.

The query generator expands the query in following different ways.

Stemming and lemmatization—The words in the user query will be stemmed and lemmatized to create original base form of the query words. Additional queries will be generated using these base forms. For example, a user query "swimming" will create a second query "swim".

Formulation of AND and OR query—First we do "AND" query for the user given keywords. If the "AND" query does not return sufficient result, we expand the query using "OR" across the user given keywords. For example, if user given query is "Racing Bicycle", then the two queries "Racing AND Bicycle" and "Racing OR Bicycle" are formed. If the query "Racing AND Bicycle" returns less than a threshold number of results (in Mobilewalla implementation that threshold value is 20), the OR query "Racing OR Bicycle" is executed. This "AND" and "OR" combination is also applied on the original base form of the user given query keywords.

Query Expansion—If the user given query does not return a threshold number of results, we expand the query using synonyms and hyponyms of the query keywords. For example, a query "Bollywood Movie" will be expanded into "Bollywood Cinema" and "Bollywood Picture", because "Cinema" and "Picture" are the synonym and hyponym of the word "Movie".

4 Scoring computation

In Mobilewalla (MW), one of our key contributions is a set of analytics that provide practically useful intelligence regarding the mobile app ecosystem. While we compute a number of such artifacts (such as Hot App, New App, Fast Moving App), two key analytics that drive most others are (a) the Mobilewalla Score (MWS) and (b) the Mobilewalla Developer Score (MWDS).

4.1 Mobilewalla Score (MWS)

The Mobilewalla Score (MWS) is a numerical value, between 0 and 1, that is computed for every app in the MW system, for a given platform. The MWS values are representations of how successful, i.e., how "hot" an app is at a given time, on a given platform, with higher values indicating "hotter" apps. In other words, an Android app with a MWS of 0.9 is considered more successful than an Android app with a MWS of 0.7.

All app stores have some way of indicating app "success". Yet, for users, these methods are not useful (in fact, quite confusing), for the following reasons:

(1) It turns out that not only do app stores have "hotness" metrics, they actually have several such measures, which often conflict, providing very unclear signals to the user. Table 3 below outlines the various app success metric attributes available in the Apple, Android, Blackberry and Windows 7 stores. The problem arises from the fact that these attributes often provide conflicting signals. Take the app "Always Up!" (AU) By AlphaWeb Plus LLP in the apple app store for example. At the time of writing of this paper, AU appears in the top 10 list of the Games-Arcade category. However, when we turn to user ratings, another key metric widely used by consumers, we find that AU has very few ratings only. Indicating high likelihood of the fact that AU is yet to find wide distribution among users. Further analysis reveals that this is not surprising, given that AU was launched only on November 2011 it is too new to have garnered much user attention or popularity. In other words, the two important "hotness" attributes in the apple app store, namely rank and ratings are glaringly conflicting in this (it turns out that such examples are very common, see the app "Wonga" and "Fish4jobs" which appears in the Top 10 in categories "Finance" and "Business" respectively. Both the apps were released on early October and November 2011 and has very few ratings).

Table 3 Important metrics for various app stores

App store	Metrics
Apple	Apple store rank, rating stars and rating counts (current and all versions)
Android	Android rank, rating stars and rating counts
Blackberry	BB store rank, rating stars, NumReviews (from users)
Windows 7	Win7 store rank, rating stars, rating count

(2) A huge problem in app stores is that the published metrics themselves are quite suspect in their value to users. For instance, consider the rank attribute in the Apple store. The importance of this attribute is undeniable, as it controls the order in which apps are made visible to the users—the expectation is that it captures how popular apps are in the market, analogous to the "best-seller" or "billboard chart" features well known in other popular consumer spaces like books, movies or music. Yet, rank clearly is not driven solely by how popular an app is—otherwise how could an app like "Wonga" and "Fish4jobs" (described above), achieve a top 10 rank within a few days of its launch, with very few rating for the apps? Evidently, app ranks embody the same bias as search engine rankings where it is well known that placement rankings of links in search results are quite influenced by purely commercial considerations. This creates a problem when an individual consumer is interested in searching for popular puzzle games or a health care company, interested in finding optimal placement for its mobile ads, wants to discover the most popular health apps. There are no ways to do this. In these, and most other use cases, app store ranks are not very trustworthy.

(3) The store provided metrics do not discriminate across the various categories and subcategories, making comparisons across categories impossible. However, users need to make such judgments. To see this, consider the following example: a large youth targeted clothing and accessories store wishes to advertise on the top apps across the "games", "lifestyle" and "entertainment" categories. Their advertising budget is limited, requiring them to find the top 15 apps across these categories. Based on store provided metrics, the best they can do is pick the top 5 apps in each of the 3 separate categories. However, this method will almost surely not yield the desired outcome, i.e., to pick the top 15 apps overall—it might very well be that the 15th ranked app in games is way more popular than the top ranked app in lifestyle (case in point: in Apple's app store, there are over 40,000 unique games while there are only 5000 unique lifestyle apps). It is clearly desirable, therefore, that there exist ways in which apps can be compared across categories—not just inside categories.

All of the above point out a glaring hole in current app stores—the lack of a unifying metric that allows users to perform a consistent, unbiased and across the board comparison of apps. In MW, we introduce the MWS metric as the industry's first such measure. The way we compute MWS, needless to say, is one of our core contributions.

Intuitively, we gather a number of available "signals" regarding the "hotness" of apps as mentioned in Table 3. From these signals, we extract those that are the most significant (mathematically, we remove auto-correlations). Based on this "core signal set" we create a mathematical model for MWS—this model, simply speaking, is a formula that combines these signal variables to output a single number between 0 and 1—the MWS value of an app. The MWS formula is presented below omitting the details for the sake of simplicity. Though the formula below will give an idea on the basis of MWS computation, the actual MWS computation is little different that includes lot of other sources such as Twitter, Facebook, Youtube and Blog postings related to an app. Due to intellectual property issues, we refrain from presenting the exact MWS compuation in this paper.

$$P_c^a = \frac{(NP_c - R_c^a)}{NP_c}, \quad \forall c \in C, \forall a \in A \tag{1}$$

$$EP^a = \max_{\forall c \in C} P_c^a, \quad \forall a \in A \tag{2}$$

$$CR^a = \left(\sum_{v \in V^a} S_v^a \times U_v^a \right) / (T^a + 1), \quad \forall a \in A \tag{3}$$

$$ER^a = \frac{\ln(CR^a+1) - ln(\min_{\forall b \in A}(CR^b)+1))}{\ln(\max_{\forall b \in A}(CR^b)+1) - \ln(\min_{\forall b \in A}(CR^b)+1)}, \quad \forall a \in A \tag{4}$$

Table 4 Notation

Symbol	Meaning
C	Set of all categories
A	Set of all applications
A_d	Set of all applications developed by developer d
V^a	Set of all versions for an application a
N_c	Number of applications in a category c
NP_c	Number of applications whose primary category is c
R_c^a	Rank of an application a in a category c. If the application a is not ranked by the store (i.e. more than 241 in AppStore, $R_c^a = N_c/2$
S_v^a	The number of stars for an application a for a version v
U^a	The number of ratings received by the application a for all versions
U_v^a	The number of ratings received by the application a for version v
T^a	The number of months the application a is in the store starting from the first version

For apps with ranking,

$$MWScore^a = 0.7 \times EP^a + 0.3 \times ER^a \quad (5)$$

For apps without ranking,

$$MWS^a = ER^a \times \min_{(\forall b \in A, \exists b \in c \& a \in c, \forall c \in C)}(MWS^b) \quad (6)$$

In Eq. 1 we compute the percentile rank P_c^a, which is a value between 0 and 1, where higher value indicates the app a is highly ranked in the category c in the native store. The Eq. 2 computes effective percentile rank EP^a, the maximum percentile rank of an app a across all categories it belongs to. Equation 3 computes the combined rating CR^a per month of an app a. ER^a computed in Eq. 4 is the normalized effective rating of an application a appropriately scaled by logarithmic function and normalized across all apps. Finally, in Eq. 5, we compute the Mobilewalla score (MWS) of app a by combining the normalized effective rating (ER^a) and effective percentile rank (EP^a) with appropriate weightage decided based on experimental and statistical analysis. For unranked apps, we do not have any EP^a, so we compute the MWS by appropriately scaling the lowest score obtained by a ranked app in the same category as the app a, by the normalized effective rating ER^a. Refer Table 4 for the meaning of notations used in Eq. 1–7 for MWS and MWDS computation.

4.2 Mobilewalla developer reputation score (MWDS)

The Mobilewalla Developer Reputation Score (MWDS), is a numerical value between 0 and 100, awarded to each developer, that indicates how "good" the developer is on a given platform. In other words, a developer on the apple platform with a MWDS of 85 is regarded as having "better" reputation than a different apple developer having a MWDS of 65.

Generally speaking, MWDS attempts to explicitly capture the "manufacturer" reputation rankings implicitly associated with most consumer products. For instance, when we decide which movie to watch, the reputation of the film director is an important input parameter. Or when we purchase a book, the reputation of the author plays a key role. This occurs because there is an expectation that a "manufacturer" who has been successful before is more likely to be successful again, compared to one with a lesser history of success.

We tested this theory in the mobile app market, and found that there is strong statistical support of it—among developers with multiple apps, prior success is strongly correlated with observed "re-success". Hence, the knowledge of developer reputation rankings is extremely helpful when looking for apps. However, there is one major difference between mobile apps and other consumer products that make the acquisition of such knowledge very difficult in the case of the former.

In virtually every consumer space (movies, cars, shoes, electronic devices, etc.) both the number of products and the number of manufacturers is "relatively" small. For instance, worldwide, at any given time, there are less than 500 directors making movies, or fewer than 250 manufacturers making smart phones. At scales like this, reputation rankings are implicitly developed and maintained. For example, pretty much any informed movie watcher in the US (and abroad) would consider the Coen brothers and Steven Spielberg to be moviemakers with high reputations, much in the same way as Versace and Louis Vuitton are considered reputed developers of women's handbags. Such reputations are developed implicitly, but are quite well known—these reputations also figure prominently in the consumption decisions of users.

For mobile apps, unfortunately, such implicit evolution of manufacturer reputation is impossible, due to the scale of the app market. Consider the fact that there are currently over 225,000 mobile app developers globally, contributing to a market of near 900,000 apps at this time, expected to cross 1 million in the next few months. Moreover, the high category fragmentation of the app market (apple itself has over 35 category segments) makes it impossible to keep track of implicit reputations. For instance, consider a user wishing to search for a utility app such as an alarm clock—it is impractical for this user to know who the most reputed alarm clock developers are (there are over 250 alarm clocks apps out there).

It is with this problem in mind that we envisioned the creation of an "explicit" developer reputation parameter—MWDS is the result. Effectively, MWDS provides app users the same ability to discriminate and choose that is commonly available in every other consumer choice scenario, by making explicitly available a critical decision variable—the reputation of the manufacturer.

For app developers themselves, MWDS promises to be a powerful marketing tool. Clearly, in the hyper-crowded app developer ecosystem, it is critical to stand out. The MWDS value for developers is a great way of doing that.

The intuition behind MWDS computation is as follows: Let us take John the developer. To compute John's MWDS,. we consider all apps developed by him and take into account the Mobilewalla App Scores (MWS) for these apps, over time. We then mathematically combine these items to produce MWDS. Leaving all the small details, the MWDS formula can be

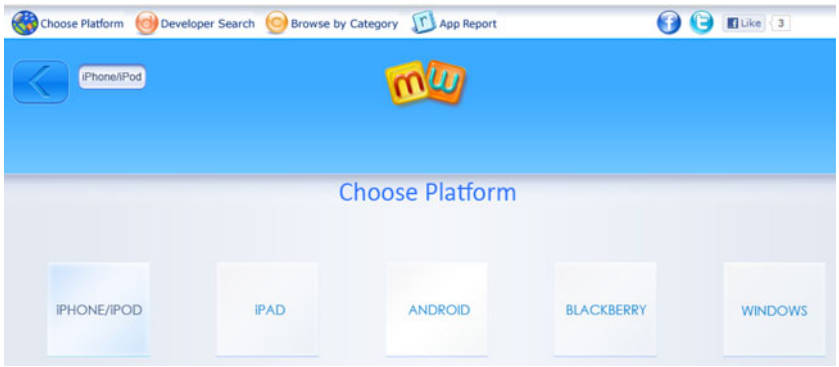

Fig. 3 Platform navigation page

presented by Eq. 7, which is the average score of 70 % top apps (based on MWS) written by the developer d.

$$MWDS_d = Avg_{(a \in A_d \cap \text{ top 70 \% of } A_d)}(MWS^a) \tag{7}$$

5 Screen shots and descriptions

The Mobilewalla architecture is flexibly implemented using a JSON interface. The application server provides a set of JSON APIs that can be invoked by any client over HTTP. Currently, the clients supported include iOS devices (i.e., iPhone/iPod/IPad), Android devices and desktop web applications. All these clients communicate with the Mobilewalla server application using the same JSON API set, but differ in the user interface offered to the end user. We will now proceed to demonstrate some of the important functionalities of the Mobilewalla application by using desktop web application as an example (the user may interact with this application at www.mobilewalla.com).

When the user arrives at the Mobilewalla application, the first step is to choose a platform of interest, i.e., the user must specify which smartphone platform is of interest to the user—iPhone/iPod, iPad, Android, Blackberry or Microsoft (the user may also choose a "don't care" option, marked as "All" in Mobilewalla) as shown in Fig. 3. Once a platform is chosen the user will be directed to the main "splash page" shown in Fig 4. In the screenshot shown in Fig. 4, the platform chosen appears on the extreme right of the top menu bar (iPhone/iPod in this case). This means the all apps presented to the user will correspond to the iPhone/iPod platform until this is explicitly changed, by selecting the "Choose Platform" widget present on the extreme left of the top menu bar.

From this screen, the user may choose to navigate the app world in a number of ways. The first, and the most common method of interaction is by entering a search query in the keyword input box. Let's assume the user enters the search term "`earth photo`". Mobilewalla returns a set of apps that fit the user's

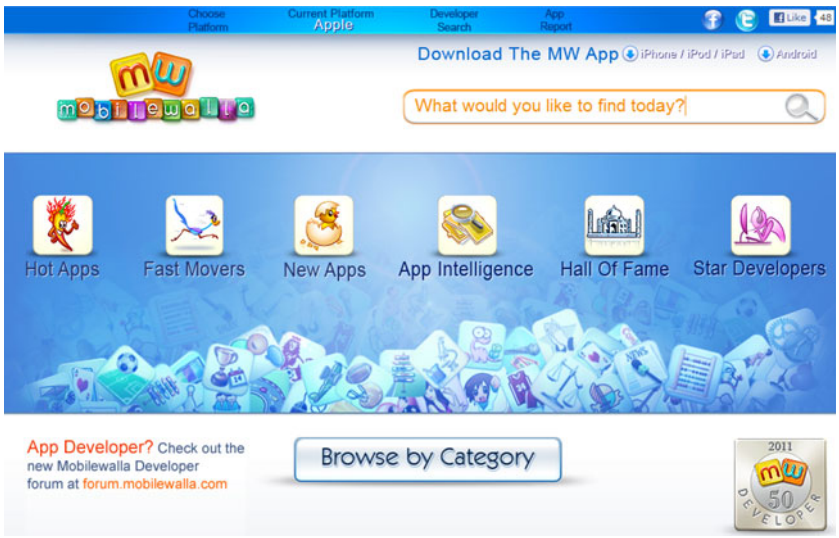

Fig. 4 Main menu page

Fig. 5 Keyword search results page

interest as shown in Fig. 5—in this view Mobilewalla provides not only the app name, but also a number of other core features such as author and price. One notable feature of this view are the *relevance* and *Mobilewalla meter* (*MW Meter*) indicators present in each app box. Relevance indicates the quality of "fit" of that app with respect to the input search query, whereas *MW Meter* is an encapsulation of the "goodness" of the app as measured by Mobilewalla (this is based on the Mobilewalla Score metric described earlier). Also, while not shown the screenshot, we also segment the apps by Free and Paid and allow a number of options to sort the result set (the user may view these by visiting mobilewalla.com).

The user may choose any app from the app-list view just described and delve into its details. Let us assume

Fig. 6 App details page

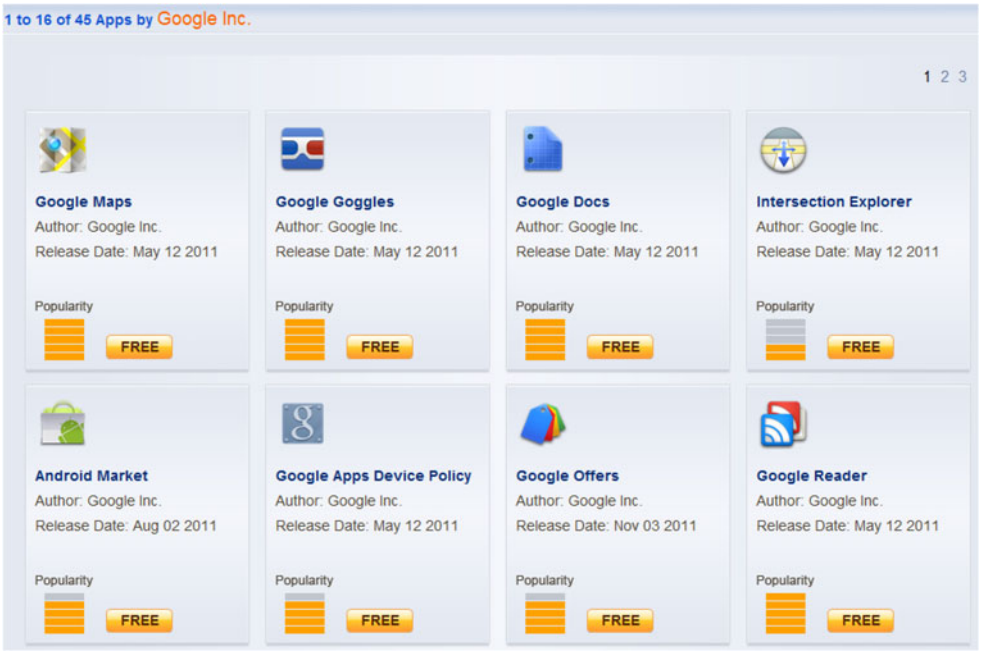

Fig. 7 Apps by developer page

the user chooses the Google Earth app. In this case she will be presented with the detail view of this app, shown in Fig 6. In this view, Mobilewalla displays details such as the app description and screenshots and also allows the user to view a number of other interesting artifacts related to this app, such as "Apps by Author" (other apps created by the author of the app detail being viewed), "Mobilewalla Score"(the Mobilewalla score history related to this app over the past 14 days), "Comments", and "Similar Apps" (similar to the "if you like this, you might also like" feature in Amazon). The screenshots corresponding to the "Apps by Author" and "Similar Apps" for the app Google Earth are shown in Figs. 7 and 8. The above two paragraphs describes how a user might interact with Mobilewalla by performing a keyword search and then drilling down on the results. However, keyword search is just one of many ways that the user can navigate Mobilewalla. He might also choose to view apps by categories, or choose one of the many "pre-defined" list options such as "Hot Apps", "Fast Movers" and "New Apps". Choosing the "Browse my category" option reveals a number of

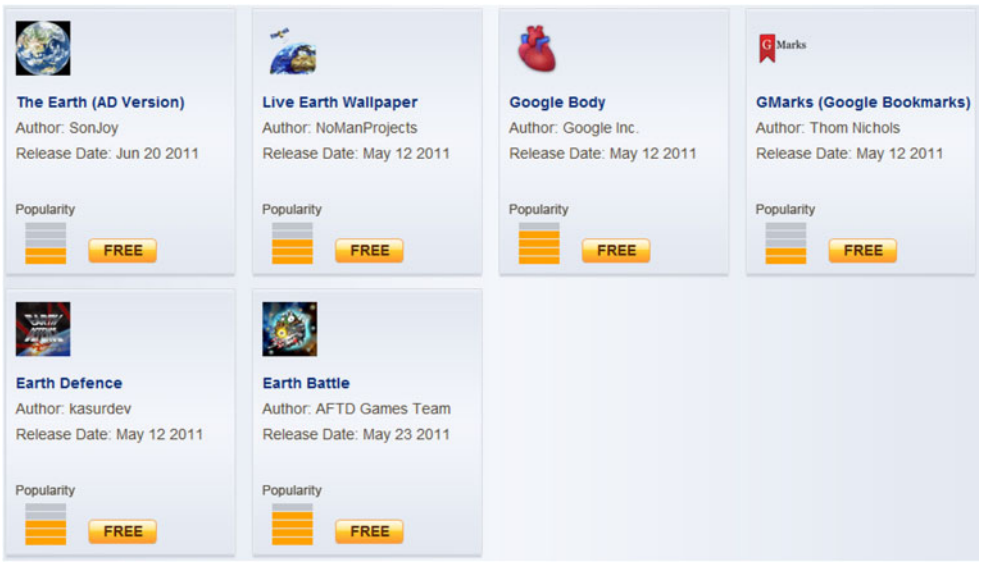

Fig. 8 Similar app page

Fig. 9 Category search results page

category icons from which the use may navigate the app world—Fig. 9 shows the results of choosing the "Maps & Navigation" category.

Similarly choosing "Hot Apps" displays the list of the top 1,000 apps ordered by their Mobilewalla Scores (Fig. 10), while "Fast Rising" apps are those whose Mobilewalls scores have demonstrated the steepest ascent, i.e., apps getting hot the fastest (Fig. 11). "New Apps" are those that are less than a month old (Fig. 12). In every case a number of sort options are available that allow users to manipulate the result set along various dimensions.

While Mobilewalla has a number of other interesting features, it is infeasible to describe them in this paper due to length restrictions. We invite the user to visit the site.

Fig. 10 Hot apps page

Fig. 11 Fast rising apps page

6 Related work

Mobilewalla is one of the earliest entrants in the app search and discovery space. In this section, we describe few alternatives available for app search.

Appolicious Inc, associated with Yahoo Inc [33], is a web application to help users easily find mobile applications that are useful and interesting to them. The Appolicious recommendation engine determines what apps you have, what apps your friends and the rest of the community have, and uses individual app ratings and reviews to suggest new apps for your use. Unlike Appolicious, Mobilewalla depends on its own developed scoring mechanism. Users can search for an app in the Mobilewalla system using keywords, which is based on its own index mechanism. Mobilewalla also identifies similar apps based on content than the usage, as is the case in Appolicious. The most impor-

Fig. 12 New apps page

tant attraction of Mobilewalla [34] is the availability of different kind of Search functions which are 'keyword based search', 'category search' and 'developer search' based on the semantics and topics. Because of this uniqueness, through Mobilewalla one can find a much larger variety of apps.

Chomp is an app search engine [35] specifically for iOS platform (iPhone and iPad), that has been acquired by Apple recently. Without any publicly available information regarding Chomp's search mechanism, the only resort we had is qualitative comparison of Chomp's search result vs. Mobilewalla search result. In Table 5, we have presented the search results for both Chomp and Mobilewalla across 5 different keywords.

It is very clear from the Table 5, that Mobilewalla is able to provide relevant results in all cases, whereas Chomp could not provide any result for 3 cases. Generally, the qualitative comparison of the search results in Mobilewalla and Chomp across these set of keywords clearly demonstrate the superiority of Mobilewalla with respect of Chomp. These keyword search results are just the subset of keyword searches that we compared across both Chomp and Mobilewalla. We are not sure about whether Chomp does the search using semantic context of the search, but the results seems to demonstrate even if Chomp uses semantic search, its application is very limited.

Mobilewalla, on the other hand, searches apps based on the extensive semantic content of both the key-

Table 5 Comparison of Mobilewalla and Chomp search result

Comparison of Mobilewalla and Chomp Search				
Search keywords	Mobilewalla free apps	Mobilewalla paid apps	Chomp free apps	Chomp paid apps
Stanford bike	Stanford iWater	Stanford college football fans	No search results	No search results
	Stanford magazine	Stanford college basketball fans		
	Stanford ticket office	FitVideo: mountain biking		
	Zing @ Stanford	Bike doctor		
	Stanford lawyer	Unofficially Stanford		
ESPN Cricket	NDTV cricket	Cricket trivia	No search results	No search results
	iCricket—most popular cricket app	Official guide—ICC cricket world cup		
	ESPN score center	Cricket reloaded		
	Crickets	Touch cricket		
	ESPN radio	Cricket coach		
Voice over IP calling apps	Rogers hosted IP voice	GV mobile +	No search results	No search results
	Forfone—free phone calls & text	Acrobits softphone		
	IP-relay	IP vision		
	IP-caller	Caller ID ringtones		
	Easy talk—free text & phone calls	IP vision pro		
Singapore taxi	Go-taxi booking app	Singapore SMRT and taxi guide	TaxiCan! (Singapore)	Singapore SMRT and taxi guide
	Taxi booking 5.0	World taxi		SingaporeTaxi
	World taxi-fares and tips	Singapore visitor guide by feel social		
	TaxiCan! (Singapore)	Singapore tourist spots		
	Taxi Lah!	Singapore subway/MRT		
Chinese cusine	Cookbook cusine	Chinese food recipes	No search results	No search results
	Bon Pate Mediteranian cusine	LanguageVideo: basic Chinese		
	Chinese cooking	Chinese Chinese		
	Chinese word search lite	Flashcards-Chinese		
	iStart Chinese!	Surf Chinese		
Vacation deals	Ski vacation deals	Vacation cruising—save money	Ski vacation deals	No search results
	Cruise finder	Budget travel complete guide	Multichoice travel	
	Fligo.pl	101 tips for traveling on a budget	Cruise finder	
	HulaCopter Hawaii	Vacation cruising		
	Red tag vacations	Budget traveler		

words and app information available in native stores. As a result, the search outputs are vastly different. For example, the search term "ESPN Cricket" in iPhone platform returned no result in Chomp, whereas in Mobilewalla it returned total 486 apps with "NDTV Cricket", "iCricket" as the top two apps. Also, unlike Mobilewalla, which is applicable in iPhone/iPod/iPad, Blackberry, Android and Windows platform, Chomp's search is limited only to Apple and Android app stores. Chomp's trending search option is based on existing popularity of the app. Whereas Mobilewalla's trending search results are based on Mobilewalla's unique scoring mechanism, which uses other parameters indicating both the developer's and user's engagement.

AppBrain is a website [36] to search Android apps available specifically in the Google Android market, whereas MobileWalla [34] covers all the major smart phone platforms (Windows, Blackberry, Android and Apple). The main core feature of Mobilewalla system is its search functionality. It has semantic enabled keyword, category and developer based search functionalities. Keyword search in Mobilewalla is implemented in a way to find the semantically meaningful apps but not purely based on exact string match like most of the currently available app search engines like AppBrain do.

uQuery.com [37] is a mobile app search platform based on social media Facebook [38]. Users can search and find applications on the iTunes App store based on what apps friends in Facebook are using. The key difference between Mobilewalla and uQuery is in the search mechanism. Mobilewalla relies on its own metrics which is combination of both the usage and the developer's engagement with an app, whereas uQuery relies on usage by friends in Facebook. The Mobilewalla's keyword based search engine is much more extensive than uQuery. Mobilewalla's keyword search can handle semantically related keywords, keywords appearing in title, description and user comments. Mobilewalla's similar app search mechanism is based on semantic similarity of apps, rather than the usage.

In summary, the key difference between the Mobilewalla and existing app search platforms is in two fronts. First, Mobilewalla depends on the semantic description of apps to enable search based on keywords and similar apps. Second, Mobilewalla ranks the app based on a scoring mechanism that incorporate both the user and the developer's involvement with the app. As discussed above, the existing search platforms consider only the user's aspect; like Mobilewalla developer's involvement with the app and the developer's history is not considered.

Having compared Mobilewalla with some of its competitors in terms of technology, let us now focus on how Mobilewalla differs in terms of usability. At high level all the above systems work in the very similar way—there is a simple GUI with one search text box. Upon submitting the search query in the search text box, the result of the search is displayed. This basic mechanism is inspired by Google search. In AppBrain, the search result is displayed as a list, whereas both in Chomp and Mobilewalla, the result is displayed in matrix fashion. We believe the biggest usability advantage of Mobilewalla compared to its competitors is the different ways the search results can be sorted. In Mobilewalla, the results are sorted by "free" app vs. "paid" app (which is also available in Chomp). Additionally, the Mobilewalla search results can be sorted based on search relevance (which is the default sorting scheme), popularity of the apps and release dates. Considering that a list of apps is a multi-dimensional data space, this allows the users to browse the results from different dimensions. Such multi-dimensional sorting mechanism is not there in Chomp.

One of the challenges for mobile app users is finding interesting apps. The specialized lists of Mobilewalla such as Hot App, Fast Movers, New App provide easys way of finding some interesting apps. This is another usability feature of Mobilewalla compared to Chomp.

Knowing which app is good and which one is not so good is an issue for app users. Chomp and AppBrain provide users with data related to reviews and native store ratings. Whereas, at Mobilewalla, we believe app users get overwhelmed with the reviews and ratings and do not always know, how to interpret these review and ratings. So we have developed a single MWS metrics, that can be used by app users to judge the quality of an app. The metrics is being widely accepted in the USA. Currently NYTimes publishes its featured app lists along with movie ratings (provided by Neilsen) based on MWS. In addition about 300 other US news papers publish MWS based app ratings. Such a single metric to measure the quality of an app instead of multiple reviews and ratings definitely increases the usability of the Mobilewalla system than its competitors.

7 Conclusion

In this paper we have described the Mobilewalla app search engine. We have described the architecture and some of the key feature details along with MWS and developer score computation. With the help of a set of

screen shots we have demonstrated some of the functionalities of Mobilewalla. Lastly, in related work we have qualitatively compared Mobilewalla system with some of the other existing systems such as Chomp and Appbrain. This is a very initial reporting of the Mobilewalla system. In future, once the intellectual property rights of Mobilewalla has been addressed by patent application, we intend to publish the details of the similarity app computation in Mobilewalla. We also intend to do a through usability test of the Mobilewalla representation of the app data compared to other competitors, Chomp and AppBrain. This will require human subject based controlled experiment. Lastly, though MWS and MWDS have been commercially accepted by news and media community, we intend to develop a theoretical basis for these score computations. In summary, we believe this paper will create a series of other research in the area of semantic search and product ratings in the mobile app domain.

References

1. Ben G. Android market grows a staggering 861.5 per cent. http://www.knowyourmobile.com/smartphones/smartphoneapps/News/781397/android_market_grows_a_staggering_8615_per_cent.html. Accessed 13 May 2011
2. Brothersoft. Mobile apps market to reach $38 billion revenue by 2015. http://news.brothersoft.com/mobile-apps-market-to-reach-38-billion-revenue-by-2015-6089.html. Accessed 13 May 2011
3. Google Inc. Android market. https://market.android.com/. Accessed 13 May 2011
4. Apple Inc. Apple app store. http://www.istoreus.com/home.html. Accessed 13 May 2011
5. Business Insider. Why native app stores like itunes and android marketplace are bad for mobile developers. http://www.businessinsider.com/why-native-app-stores-like-itunes-and-andoid-marketplace_are-bad-business-for-mobile-developers-2011-5. Accessed 13 May 2011
6. Amit Agarwal. Would you like to try android apps before buying? http://www.labnol.org/internet/android-apps-try-before-buying/19422/. Accessed 13 May 2011
7. Gigaom. Appsfire scores $3.6m as app discovery demands grow. http://gigaom.com/2011/05/30/appsfire-scores-3-6m-as-app-discovery-demands-grow/. Accessed 13 May 2011
8. CNet. The mobile app discovery problem. http://news.cnet.com/8301-30684_3-20011241-265.html. Accessed 13 May 2011
9. Zdnet. Google finally applies its own search technology to apps. http://www.zdnet.com/blog/mobile-gadgeteer/google-finally-applies-its-own-search-technology-to-apps/3332?tag=mantle_skin;content. Accessed 13 May 2011
10. Zdnet. App search engine competition heating up. http://www.zdnet.com/blog/mobile-gadgeteer/app-search-engine-competition-heating-up/3785. Accessed 13 May 2011
11. New York Times. Popular demand: App wars. http://www.nytimes.com/interactive/2011/05/16/business/media/16most.html?ref=media . Accessed 13 May 2011
12. Lazaridis M. Blackberry app world. http://us.blackberry.com/apps-software/appworld/. Accessed 13 May 2011
13. Gates III WHB. Windows. http://marketplace.windowsphone.com/Default.aspx. Accessed 13 May 2011
14. Heydon A, Najork M (1999) Mercator: a scalable, extensible web crawler. World Wide Web 2:219–229. doi:10.1023/A:1019213109274
15. Boldi P, Codenotti B, Santini M, Vigna S (2004) Ubicrawler: a scalable fully distributed web crawler. Softw Pract Exper 34(8):711–726. doi:10.1002/spe.587
16. Hsieh JM, Gribble SD, Levy HM (2010) The architecture and implementation of an extensible web crawler. In: Proceedings of the 7th USENIX conference on networked systems design and implementation, ser. NSDI'10. USENIX Association, Berkeley, CA, USA, pp 22–22. http://portal.acm.org/citation.cfm?id=1855711.1855733
17. Garrett JJ. Ajax. http://www.ajax.org/. Accessed 13 May 2011
18. Microsoft Inc. The official Microsoft Silverlight web site. http://www.silverlight.net/. Accessed 17 May 2011
19. Adobe Inc. ActionScript 3.0 Overview. http://www.adobe.com/devnet/actionscript/articles/actionscript3_overview.html. Accessed 17 May 2011
20. Kayak. Kayak. http://www.kayak.com. Accessed 17 May 2011
21. Crockford D. Json. http://json.org/. Accessed 13 May 2011
22. Resig J et al. JQuery. http://jquery.com/. Accessed 13 May 2011
23. Apache. Apache cassandra project. http://cassandra.apache.org/. Accessed 24 Nov 2011
24. Apache. Lucene. http://lucene.apache.org/. Accessed 13 May 2011
25. Agrawal D, Das S, El Abbadi A (2011) Big data and cloud computing: current state and future opportunities. In: Proceedings of the 14th international conference on extending database technology, ser. EDBT/ICDT '11. ACM, New York, NY, USA, pp 530–533. doi:10.1145/1951365.1951432
26. Suchanek FM, Kasneci G, Weikum G (2008) Yago: a large ontology from wikipedia and wordnet. In: Web semantics: science, services and agents on the World Wide Web. World Wide Web conference 2007 semantic web track, vol 6(3), pp 203–217. http://www.sciencedirect.com/science/article/pii/S1570826808000437
27. Apache. OpenNLP. http://incubator.apache.org/opennlp/. Accessed 13 May 2011
28. Amazon Inc. Elastic block store. http://aws.amazon.com/ebs/. Accessed 24 Nov 2011
29. Princeton University About wordnet. http://wordnet.princeton.edu. Accessed 17 May 2011
30. VMWare Inc. RabbitMQ. http://www.rabbitmq.com/. Accessed 24 Nov 2011
31. Apache. Hadoop JoBTracker. http://wiki.apache.org/hadoop/JobTracker. Accessed 24 Nov 2011
32. Apache. Hadoop. http://hadoop.apache.org/. Accessed 24 Nov 2011
33. Warms A. Find mobile apps you will love. http://www.appolicious.com/. Accessed 17 May 2011
34. Datta A. Helping you navigate the app world. http://www.mobilewalla.com/. Accessed 13 May 2011
35. Ben K, Cathy E. The appstore search engine. http://www.chomp.com/. Accessed 17 May 2011
36. Appbrain. Find the best android apps. http://www.appbrain.com/. Accessed 17 May 2011
37. uQuery. The appstore search engine. http://www.uquery.com/. Accessed 17 May 2011
38. Zuckerberg ME. Facebook. http://www.facebook.com. Accessed 13 May 2011