Vietnam Journal of Computer Science (2018) 5:133–142
https://doi.org/10.1007/s40595-018-0109-9

REGULAR PAPER

Precomputing architecture for flexible and efficient big data analytics

Nigel Franciscus[1] · **Xuguang Ren**[1] · **Bela Stantic**[1]

Received: 22 August 2017 / Accepted: 10 May 2018 / Published online: 19 May 2018

Abstract
The rising of big data brings revolutionary changes to many aspects of our lives. Huge volume of data, along with its complexity poses big challenges to data analytic applications. Techniques proposed in data warehousing and online analytical processing, such as precomputed multidimensional cubes, dramatically improve the response time of analytic queries based on relational databases. There are some recent works extending similar concepts into NoSQL such as constructing cubes from NoSQL stores and converting existing cubes into NoSQL stores. However, only limited attention in literature have been devoted to precomputing structure within the NoSQL databases. In this paper, we present an architecture for answering temporal analytic queries over big data by precomputing the results of granulated chunks of collections which are decomposed from the original large collection. In extensive experimental evaluations on *drill-down* and *roll-up* temporal queries over large amount of data we demonstrated the effectiveness and efficiency under different settings.

Keywords NoSQL · Data warehouse · Precompute · Temporal

1 Introduction

With the development of data-driven applications in many aspects of our daily lives, it is worthy to praise the significance of big data whose value has already been recognized by both industry and academia. A significant amount of data is being collected and analyzed to support various decision makings, and that amount is expected to increase every year. One of the major tasks in mining big data is to answer the analytic queries efficiently. Analytic queries often involve sophisticated aggregations which demand significant computing powers. The huge volume of big data, along with complexity of these queries pose a big challenge for efficient processing. Aiming to tackle these challenges and to enhance the performance of analytic query processing, the concept of data warehouse [13] and OLAP [6] have been introduced.

✉ Nigel Franciscus
n.franciscus@griffith.edu.au

Xuguang Ren
x.ren@griffith.edu.au

Bela Stantic
b.stantic@griffith.edu.au

[1] Institute for Integrated and Intelligent Systems, Griffith University, Gold Coast, QLD, Australia

Data warehouse integrates data from different data sources into large repositories. Data arriving into data warehouses are normally in denormalise multidimensional form which will be stored into data cubes to reduce the cost of costly table joins. A large part of modern OLAP systems are built on top of data warehouses which are stored with additional information (e.g., metadata). An essential and widely used technique in OLAP systems is *precomputation* where analytic results are precomputed and materialized in the data warehouses. When a user submits queries, the system simply retrieves the corresponding pre-computed results and therefore efficiently returns the final result to the user.

Previous techniques proposed in data warehouses and OLAP systems are mainly focusing on relational data structure and relational databases. However, it is evident that relational DBMS are struggling to handle large volume of unstructured data as they do not scale well [20,24]. Moreover, relational DBMS are not well-equipped to handle the new multidimensional networks [4,23,25]. The rise of NoSQL databases [21] has attracted the attention of database community due to its flexibility in providing schema-later architecture and its scalability for handling huge amount of big data. Various NoSQL databases have been chosen and applied in many domains, which leads to more and more data being stored in NoSQL databases. Consequently, it has become an urgent need to process analytic queries based on the NoSQL databases efficiently. Some recent works are extending the techniques of data warehouses and OLAP into NoSQL. The work of [19] present strategies for constructing cubes from NoSQL stores. In contrast, the work in [5] proposes method to convert existing cubes into NoSQL stores. However, there are few works focusing on the precomputing structure dedicated to NoSQL databases.

For many modern big data applications, the complexity of analytic tasks often require the combination of NoSQL databases and Hadoop. NoSQL databases have been known for its schema-less design that flexibly translate any data into the desired format while Hadoop fills the gap of scalability with its MapReduce framework. These two platforms work in conjunction to achieve a large-scale interactive real-time processing. Hadoop ecosystem is designed as a highly fault-tolerant system for batch processing of long-running complex jobs on very large datasets. Due to the lack of interactive exploration in HDFS, often NoSQL databases are used as the output sink for the MapReduce process for real-time interaction. To speed up the computation even further, an efficient pre-computation has becoming a good alternative [10].

Motivated by the above findings, in this paper we present an architecture for answering temporal analytic queries over big data within NoSQL database, in particular document-oriented and key-value store. As the time is an essential dimension for most of data analytic platforms, we choose temporal queries aspect as focus and as the starting point of this work. Queries on this specific part of the data (e.g., timestamp) are costly, often requiring full scan of all key-value pair despite for simple equality queries [17]. We plan extend our work into other dimensions in future works. The basic idea of proposed architecture is to divide the original data into separated and smaller chunks and then precompute the results for each chunk. The precomputed results are then materialized in the NoSQL database. We process the upcoming analytic queries based on the precomputed results.

1.1 Contribution

This paper is the extended version of the precomputing architecture for answering analytic queries for NoSQL databases [11]. We extend our previous work with further practical evaluations of the *drill-down* and *roll-up* temporal queries over a large amount of data in the application perspective, specifically:

1. We proposed the technique to index raw temporal data into separated and smaller chunks based on temporal interval.
2. We designed efficient storage structures for the precomputed results in the document-oriented and key-value databases.
3. We answered three types of common query models along with the strategies to answer each query type.
4. We conducted extensive experiments to demonstrate the performance of proposed architecture in the real world end-to-end applications.

1.2 Organization

The rest of the paper is organised as follows: in Sect. 2, we give some related works; in Sect. 3, we present the details of our precomputing architecture; in Sect. 4, we provide the experiment results; and finally in Sect. 5 we conclude the paper and indicate some future work.

2 Related work

In this section, we present some related work which we classify into two main categories.

(1) *NoSQL database.* According to a survey presented in [12] there are more than 100 NoSQL databases developed for various purposes. Specifically, No-SQL databases can be classified into four classes:

(a) *Key-value* stores the data as key-value pairs where the value can be anything and is treated as opaque binary data, the key is transformed into an index using a hash function. Redis is one of the widely used key-value databases.
(b) *Column-family* applies an column-oriented architecture which is contrast to the row-oriented architecture in RDBMS. Cassandra and HBase are two most used column-family databases.
(c) *Document-database* treats the document as the minimum data unit and is designed deliberately for managing document-oriented information, such as JSON, XML documents. MongoDB is a typical document database which is designed to handle JSON documents.
(d) *Graph-database* models the data as graphs and focuses more on the relationships between data units. There are over 30 graph database systems such as Neo4j, Titan, and Sparksee.

In this work we stored data in two NoSQL databases, MongoDB and Redis, which have different pros and cons and are using entirely different mechanisms. However, we present the query processing performance for those two databases based on our pre-computing structure.

(2) *Data warehouse and OLAP.* The concept of data warehouse and OLAP have been proposed very early aiming to answer analytic queries efficiently. The key structure in data warehouse is the cube which is normally stored as a denormalised multidimensional table in relational databases [3,7]. A large part of modern OLAP systems are built on top of

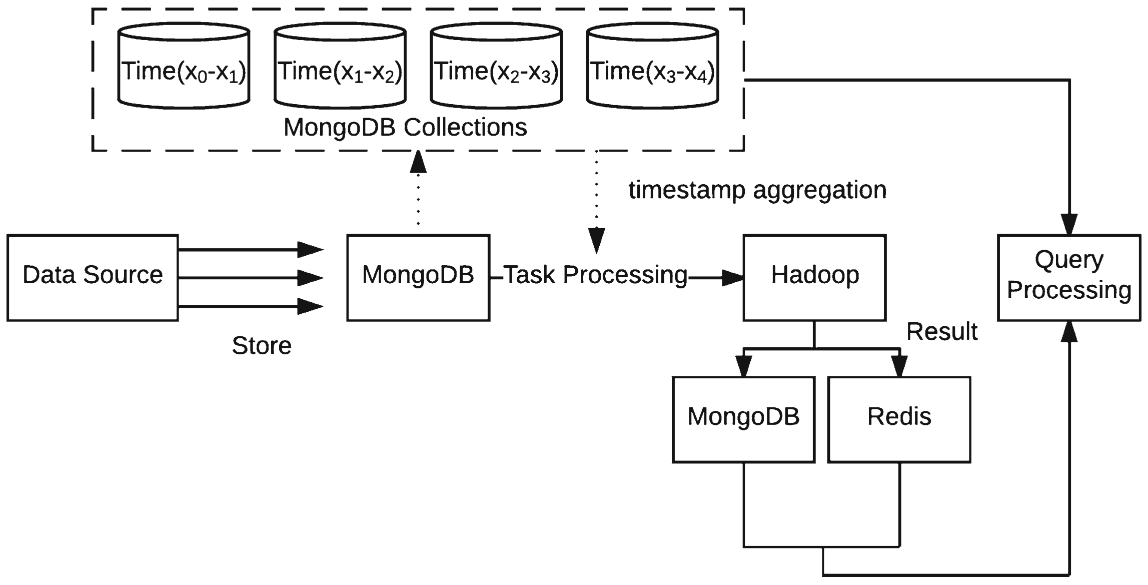


Fig. 1 Pre-computing architecture

data warehouses and utilize the cubes when processing analytic queries [8] or time-range queries [22]. Some advances have been made to extend OLAP into emerging data, such as imprecise data [2], taxonomies [18], sequence [15], and text [14]. There are some recent works extending the techniques of data warehouses and OLAP into NoSQL. The work of [19] present strategies for constructing cubes from NoSQL stores. In contrast, the work in [5] gives the rules in converting existing cubes into NoSQL stores. However, only few works studying the precomputing structure deliberately within NoSQL databases.

In contrast to previous work, we focus on the processing of analytic queries for NoSQL databases where no data are stored in relational databases. We propose an index structure suited to answer *drill-down* and *roll-up* queries over large amount of data within fast response time. Similar to work in [22], we particularly focus on the temporal queries with the time-range as the query parameter.

3 Precomputing architecture

In this section, we present design of the precomputing architecture. We first give an overview of the architecture, then we elaborate each component, (1) Raw data indexing, (2) Precomputed results structure and (3) Query answering.

3.1 Overview

The overview of proposed precomputing architecture is shown in Fig. 1. It can be divided into several inter-related components: (1) *Raw data indexing*, where we collect the raw Twitter data and store them into a NoSQL database (MongoDB) as time-indexed collections. The dotted line represents the temporal indexing structure in MongoDB where the preprocessed results are suited to the task processing, for example, mapreduce. (2) *Precompute results structure*, where we execute analytic jobs (MapReduce based on Hadoop) and then store the precomputed results into NoSQL database (MongoDB[1] and Redis[2]). (3) *Query answering*, where we apply efficient strategies to answer queries by utilizing the precomputed results through merging. As a case study, we demonstrate our architecture using specific data sources, database platforms, analytic jobs and processing techniques in this paper, as indicated within the above parentheses. However, it is worth noting that our architecture is quite flexible and can be easily extended to other use cases. We present more details about each chosen specific ingredient.

Data source As shown in Fig. 1, we use Twitter as the data source as a case study. Twitter is an online social networking service that enables users to post short 140-character messages called "tweets". Twitter is widely used in monitoring society trends and user behaviors due to its large user pool [1]. The tweets are naturally formatted into JavaScript Object Notation (JSON) and they include the textual content as well as the posted time. Every tweet consists of several attributes embedded as the property beside the main text (Fig. 2). We test the performance for Twitter dataset both with the attributes and without the attributes.

[1] MongoDB, https://www.mongodb.com/.

[2] Redis, https://redis.io/.

```
{
    "_id" : ObjectId("56d3f9c73de7da50a2be5754"),
    "truncated" : false,
    "created_at" : "Mon Feb 29 07:57:15 +0000 2016",
    "in_reply_to_user_id" : null,
    "coordinates" : {
        "type" : "Point",
        "coordinates" : [
            153.43526,
            -28.05122
        ]
    },
    "in_reply_to_status_id_str" : null,
    "retweet_count" : 0,
    "is_quote_status" : false,
    "possibly_sensitive" : false,
    "filter_level" : "low",
    "text" : "From where I love to be. #goldcoast2016 @ Turtle Beach
              Resort https://t.co/5vG16e4qbf",
     "favorite_count" : 0,
    "id" : NumberLong("704213829782216704"),
    "retweeted" : false,
    "timestamp_ms" : "1456732635337",
    "place" : {
        "country" : "Australia",
        "id" : "017453ae077eafd3",
        "place_type" : "city",
        "name" : "Gold Coast",
        "attributes" : {
        },
    "lang" : "en",
    "favorited" : false,
    "user" : {
  },
    "contributors" : null
}
```

Fig. 2 Tweet structure

Database platform MongoDB is an open source NoSQL document-store database, founded by 10gen. MongoDB stores data in document layout consists of field-value which can be nested as an associate arrays. Documents are serialised in JSON and written internally as Binary JSON (BSON). The Twitter data is in JSON format which is generally supported by the MongoDB. We choose MongoDB to store the raw data in our case study due to its in-house flexibility. MongoDB provides some features from relational database management system like sorting, compound indexing and range/equal queries. Additionally, MongoDB has its own aggregation capability and in-house MapReduce operation. One of the major drawbacks from MongoDB is that it does not guarantee concurrency which limit the native MapReduce program from running in multi-thread. This is due to the implementation of SpiderMonkey JavaScript engine, known for its threadsafe. In addition, MongoDB in-house MapReduce has poor analytic libraries compared to Hadoop. This leads us to use Hadoop in conjunction to provide better MapReduce computation.

We also store the end result in Redis database to compare the end-to-end performance with MongoDB. Redis is a fast open-source key-value store that can instantiate result from Hadoop computing platform. Since keys may contain strings, hashes, lists, sets and sorted sets, Redis can be used to support the final metrics for front end visualisation to serve data out of Hadoop, caching the 'hot' pieces of data in-memory for fast access. Combining this simple client with the power of MapReduce will let you write and read data to and from Redis in parallel.

NoSQL databases–Hadoop integration The MongoDB-Hadoop connector[3] is a plugin for Hadoop to integrate with MongoDB as the source and/or sink instead of HDFS. Note that we opt not to use HDFS due to the the interactivity of data exploration in MongoDB, although it is evident that reading and writing time from Hadoop to HDFS is faster compared with MongoDB [9]. Further, our main intention to index the raw data is primarily for reading data from MongoDB to Hadoop. At this time this paper is written there is no official Redis-Hadoop connector for writing the output result to Redis from Hadoop available. However, there are some open source connectors that allows the integration between Redis and Hadoop. We opt to use Jedis,[4] a Java client library to connect both platform.

Analytic jobs The precomputing architecture is designed to process data that are sequential or based on the order. Therefore, it is flexible to compute a variety of jobs for both spatial and temporal data. However, we consider the importance of temporal data since they typically have lower granularity which may consistently create excessive seek index problem.

Text aggregation is a widely used analytic job in many literatures. Its intuitive application is the word frequency which is intensively used to detect hot topics and trends in the society. Compared with word frequency, sometimes we are more interested in the frequency of word-pair (co-occurrence of two words) as it can help us to detect hidden patterns. Therefore, we choose the job of computing word-pair frequency in our case. That is given a set of tweets and any word-pair, we compute the number of tweets in which this word-pair co-occurred.

Processing techniques We choose Hadoop as the processor to execute the word-pair jobs. Hadoop is an open source implementation of MapReduce framework. As previously stated, although MongoDB ships with in-house MapReduce, it has poor analytic libraries compared to Hadoop. Additionally, Hadoop has better integration with other big data platforms with its specialised cluster management such as Zookeeper[5]. We later store the precomputed results into both Redis and MongoDB.

[3] MongoHadoop, http://api.mongodb.org/hadoop/.

[4] Jedis, https://github.com/xetorthio/jedis.

[5] Zookeeper, http://zookeeper.apache.org/.

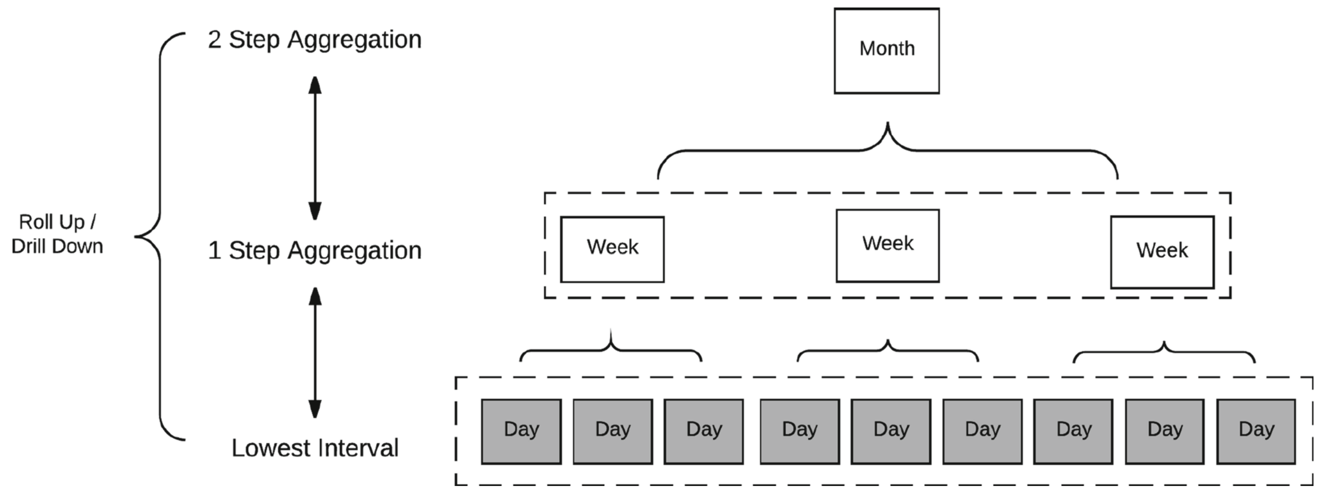


Fig. 3 Time interval index structure

3.2 Raw data indexing

It is evident that tremendous amount of data is difficult to process without proper indexing. However indexing the high-cardinality attribute, such as timestamp, is not suitable due to excessive seek [22]. For example, there will be numerous index entries if we index every specific timestamp for the tweets, which will lead to a higher latency. To tackle this problem, in this subsection, we introduce the technique of time interval index inside the collection layer of MongoDB.

Specifically, tweets are grouped into a single collection where the time of those tweets are within the same interval. The length of the interval can be tuned based on the dataset; it can be an hour, a day, or a month. We use the timestamp of this time interval as the name of the corresponding indexed collection. By utilizing the time interval index, we dramatically alleviate the cost of index seeking while still be able to support *drill-down* and *roll-up* temporal queries. Consider the example index structure in Fig. 3, we choose a day as the time interval. The tweets posted on the same day (shaded box) will be grouped into the same collection. It is worth noting that a week is a higher interval of a day; however, we do not store a separated collection to group the tweets in the same week. As this will dramatically increase the storage size.

In order to support the *drill-down* and *roll-up* temporal queries, we precompute the analytic results for each indexed collection. For example, we precompute the results for the *day* collections in Fig. 3. For each time-range query, the system will answer the query using *bottom-up* merging approach. Specifically, given a time range query whose range is more than 1 day, we first select the tweets collections within this range and load their corresponding precomputed results. Then we merge these results to get the final results to answer the query. By implementing this technique, we remove the necessity to precompute/store the result for super-interval collection such as weekly, monthly or yearly.

3.3 Precomputed results structure

As discussed in the above subsection, we precompute the analytic results for each indexed collection. In this subsection, we study the structure to store the precompute results which are the frequencies of word-pair in tweets. We present the structures for two NoSQL databases: MongoDB and Redis. For the self-completeness of this paper, we also present our MapReduce algorithms to compute the frequencies of word-pair.

MongoDB structure In MongoDB, we use a separate *collection* to store the results of each indexed collection. Each result collection contains a list of frequency results for word-pairs. The format of each frequency result for any word-pair is in the following document format:

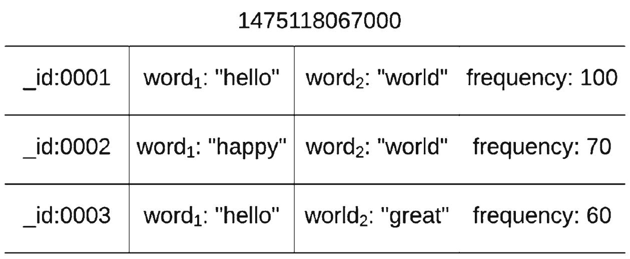
$$[\_id, word_1, word_2, frequency]$$

where *_id* is created automatically by MongoDB if not specified, $word_1$ and $word_2$ are the words in this word-pair and the *frequency* is the number of tweets in which this word-pair co-occurred. Consider the example in Fig. 4, the name of the result collection is in timestamp label *1475118067000* (29 Sep 2016). The *hello* and *world* co-occurred in 100 tweets which are posted on the day of 29 Sep 2016.

Redis structure Redis is an in-memory key-value database. We use a combination of timestamp and the word-pair as the key and the frequency as the value. The format is given as follows:

$$[time_x\_word_1\_word_2 : frequency]$$

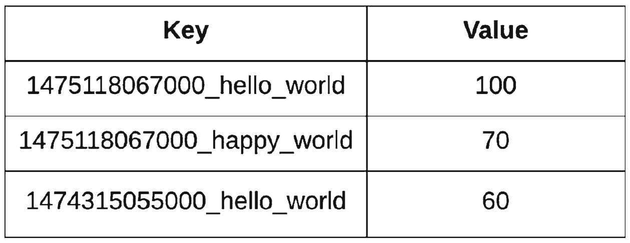
Redis support searching based on *key pattern*, thus, we can quickly lock down to the corresponding *set* of records



1475118067000

_id:0001	word$_1$: "hello"	word$_2$: "world"	frequency: 100
_id:0002	word$_1$: "happy"	word$_2$: "world"	frequency: 70
_id:0003	word$_1$: "hello"	world$_2$: "great"	frequency: 60

Fig. 4 MongoDB result structure



Key	Value
1475118067000_hello_world	100
1475118067000_happy_world	70
1474315055000_hello_world	60

Fig. 5 Redis result structure

when given a specific timestamp and/or word-pair. As we can see in the example in Fig. 5, the *hello* and *world* co-occurred in 100 tweets which are posted on *1475118067000*(29 Sep 2016) while they co-occurred in 60 tweets which are posted on *1474315055000*(19 Sep 2016).

Now we give our MapReduce algorithm for computing the frequencies of word-pairs, as shown in Algorithms 1 and 2. We take the twitter collections and a pre-fixed index interval P as input in map Algorithm 1. We first extract the timestamp T_i(Line 1) and message (Line 2) information from the tweet. Then we compute the index stamp by running a modular computation from T_i on P (Line 3). In Line 4, we tokenize and clean the text message to get a set of meaningful words. After that, for each word-pair with a preserved order, we record its appearance by 1. The reduce Algorithm 2 is straightforward to understand which sums up the frequencies for each word-pair.

Algorithm 1: $Map_{word-pair}$

```
   Input: A tweet $T_w$ within the collections, index
          interval $P$
1  Timestamp $T_i \leftarrow ExtractTime(T_w)$
2  Tokenizer $T_x \leftarrow ExtractText(T_w)$
3  IndexStamp $IndT_i \lceil\leftarrow T_i/P\rceil$
4  Tokenizer $Tok_x \leftarrow TokenizeClean(T_x)$
5  for each token $t_x \in Tok_x$ do
6      for each token $t'_x \in T_x$ do
7          if $strCompare(t_x, t'_x)$ ¿ 0 then
8              $Key \leftarrow T_i$- $t_x$- $t'_x$
9              context.write($Key$, one)
10         end
11     end
12 end
```

Algorithm 2: $Reduce_{word-pair}$

```
  Input: $Key$, List $values$
1 $sum \leftarrow 0$
2 for each value $val \in values$ do
3     $sum$ += $value$
4 end
5 context.write($Key$, $sum$)
```

The complexity of the MapReduce algorithm is $O(N \times M^2)$ where N is the number of tweets and M is the number of meaningful word in each tweet.

3.4 Query answering

In above subsections, we presented the indexing strategy and the structures to store the precomputed results. Now we are ready to study the process of answering user queries. We classify the user queries into three types: (1) Single selectivity query, (2) Drill-down query, (3) Roll-up query as we can see in the following models:
1. Single selectivity query
QUERY data WHERE $time = T_x$ WITH $Gra(T_x) = \Phi$.
2. Drill-down query
QUERY data WHERE $time = T_x$, $time = T_y$ AND T_y-$T_x < \Phi$ WITH $Gra(T_x, T_y) < \Phi$.
3. Roll-up query
QUERY data WHERE $time = T_x$, $time = T_y$ AND T_y-$T_x > \Phi$ WITH $Gra(T_x, T_y) > \Phi$ OR $Gra(T_x, T_y) = \Phi'$ IF $Gra(T_x, T_y) = \Phi'$.

In the above models, we use Φ to denote the interval when we index the raw data (as mentioned in Sect. 3.2). The function $Gra(T_x, T_y)$ is to decide the granularity of the parameter time T intuitively, for example, Gra (12AM 15 Sep 2016) = $hour$ and for Gra (15 Sep 2016) = day. Accordingly, *Single Selectivity query* aims to query the data falling into a single indexed data collection. *Drill-down query* aims to query the data which are a subset of a single indexed collection. *Roll-up query* aims to query the data involves multiple indexed collections. We use the parent interval collection Φ' when the interval given in the roll-up query matches T_y-$T_x = \Phi'$. Consider the following example where each one query corresponds to one query type respectively.
1. Word-pair frequency on 02/April/2016.
2. Word-pair frequency from 9:00pm of 08/April/2016 to 11:00pm of 08/April/2016.
3. Word-pair frequency from 18/April/2016 to 28/April/2016.

1. The time is trivial to answer the single selectivity query, as we only need to navigate to the corresponding result collection of MongoDB (set of records of Redis) by the timestamp.
2. To answer the drill-down query, we need to navigate to the corresponding indexed data collection, fetch the tweets falling into the time range and then execute the word-pair job onto the filtered tweets. The time of this process depends on

the complexity of the analytic job to be executed and can be very slow if size of the fetched tweets is large. 3. To answer the roll-up query, we need to merge multiple result collections in MongoDB (sets of records in Redis) falling into the time range. This process is similar to the table-join in the relational database. Since the weekly interval partially covers some of the daily interval (18 April 2016–24 April 2016), we select the respective weekly collection and the rest of the daily collections until 28 April 2016.

Algorithm 3: MergeResults

```
Input: Multiple precomputed results $T = \{T_1 \ldots T_n\}$
Output: final result $R$
1  HashMap $H \leftarrow \emptyset$
2  for each collection $T_k \in T$ do
3      for each document $w \in T_k$ do
4          if $w.word_1\_w.word_2$ is not in $H$ then
5              $H(w.word_1\_w.word_2) \leftarrow w.frequency$
6          end
7          else
8              $H(w.word_1\_w.word_2) \leftarrow$
               $H(w.word_1\_w.word_2) + w.frequency$
9          end
10     end
11 end
12 Result $R \leftarrow JSON(H)$
13 return $R$
```

We present a basic algorithm to merge multiple result MongoDB collections in the wordpair job, as shown in Algorithm 3. As we can see in the merging algorithm, the algorithm takes multiple results as input and output the word-pair result R. A hashmap H is used to temporarily save the frequency(value) of the *word_pair*(key) (Line 1). The algorithm iterates through each collection and visits each document inside the collection (Line 2 to 10). For each document, if there is no such *word_pair* in the hashmap, we add a new *word_pair* to the hashmap (Line 4 to 6). If there is already one, we just add up the frequency (Line 7 to 9). The above algorithm can be very fast if we tune the index interval properly. The merging algorithm for Redis is similar to Algorithm 3, we omit it here.

It is worth noting that a larger interval leads to a larger document collection. Many queries will fall into the drill-down type. When the number of tweets in one indexed collection is large, it will increase the time to answer a drill-down query. In contrast, a smaller interval will lead to many result collections(sets). Many queries will fall into the roll-up type. Excessive merge will increase the time to answer a roll-up query. Therefore, it is a trade-off between the performance of drill-down and roll-up when tuning the index interval.

4 Experiments

Our architecture has already demonstrated its effectiveness within a practical HumanSensor project [11]. In this section,

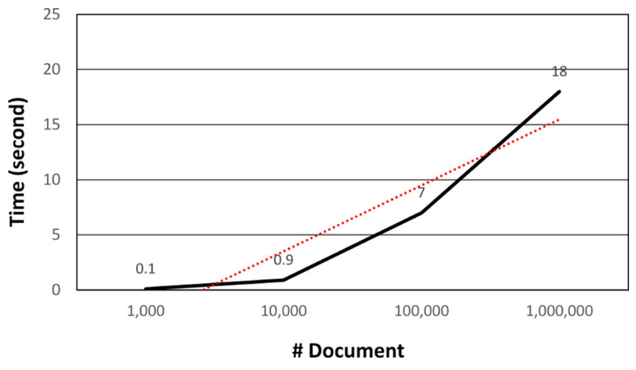


Fig. 6 Document level processing time

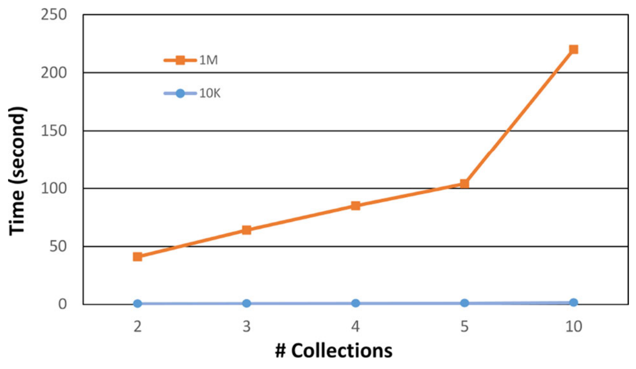


Fig. 7 Collection merging

we present our experiment results so as to study the response time of the query answering under different data settings. We describe the result of the core processing inside the database and the end-to-end processing time from back-end to front-end.

4.1 Dataset and environment

The twitter data in our experiment were downloaded though the public API provided by Twitter. We wrapped 5 datasets which contains 200×10^3, 400×10^3, 600×10^3, 800×10^3 and 1 million tweets, respectively. For each dataset, we indexed data according to a *day* interval. Bigger dataset will lead to more indexed collections and more documents within each collection. We synthetically generated three query sets for each query type, each of which contains 100 queries. The process of answering selectivity query and roll-up query utilized the precomputed results in MongoDB and Redis, thus we report the average response time of these two types for MongoDB and Redis respectively. As drill-down query only involves indexed data collections which are stored in MongoDB, we simply report the the average response time for it.

Our experiments were conducted on a cluster with 20 nodes, each node is equipped with quad core Intel(R) Core(TM) i5-2400 CPU @ 3.10 GHz with 4GB RAM. We

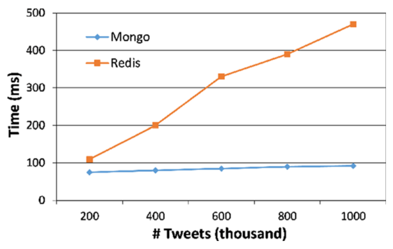


Fig. 8 Single selectivity

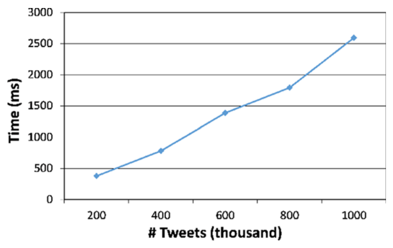


Fig. 9 Drill-down

used Hadoop (version 2.6.0), MongoDB (version 3.2.9) and Redis (version 3.0.1).

4.2 Results and analysis

Figure 6 depicts the processing time to read a single collection in MongoDB. We measured different number of documents varying from thousand to million level. The result shows that MongoDB performs scan in a linear scale. For ten thousand documents, it takes approximately a second to read and for a million it takes up to 18 s. To compute the time range, we merge the result for each affected collection. To merge multiple collections, we use a join function which takes the value of each key, mapping them into hashmap, then group into final result. Figure 7 represents the multi-collection processing time. We calculate two different size of collections, one million and ten thousand respectively. For each collection involves in the merging process, the time takes to compute is increasing linearly. For collections at a million document level, the merging cost takes approximately 6 s per collection. However, when the number of document per collection is small, the merging cost is trivial. In practical, keeping low number document will benefit the overall processing. Results suggest that dividing large collection into several partitions greatly benefit the entire processing time.

The results of processing single selectivity query are given in Fig. 8. As we can see, the time cost of answering selectivity query almost keeps constant if precomputed results are stored in MongoDB. While it depicts a linear increment when utilizing the precomputed results stored in Redis. The reason for this phenomena is due to the storage structure of results in MongoDB and Redis. For MongoDB, we saved the results of an indexed data collection in a separated collection. Given

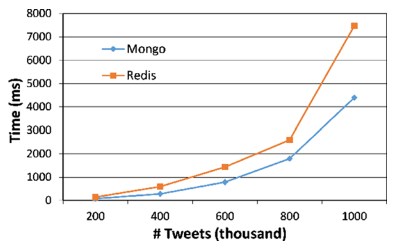


Fig. 10 Roll-up

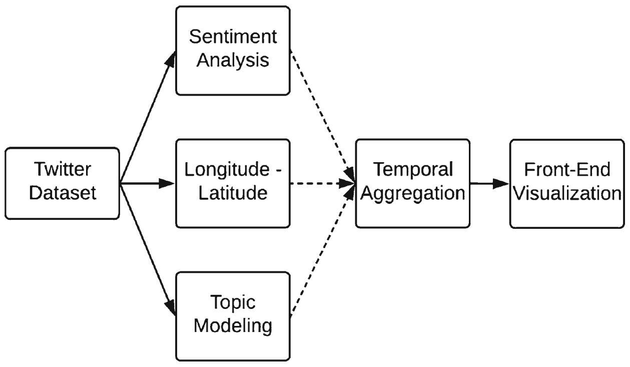


Fig. 11 End-to-end processing tasks

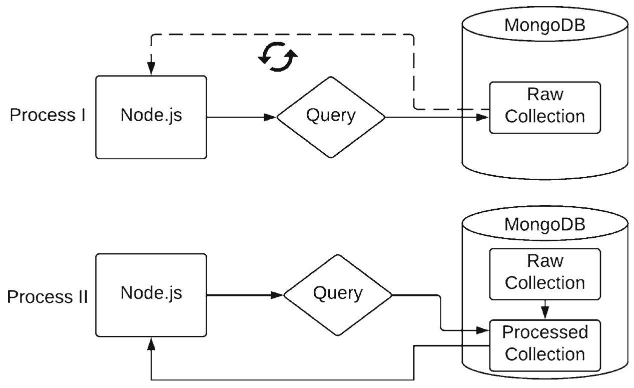


Fig. 12 Front-end vs back-end processing

a result collection name, the time to locate the corresponding collection is a hash-search which are trivial and almost constant. While the results of an indexed data collection for Redis are spread into the KEY. The internal pattern search of Redis takes a linear time in terms of the number of KEYS.

Figure 9 presents the results of answering drill-down query. As discussed in Sect. 3.4, the time cost by answering drill-down query depends on the size of the indexed data collection and the complexity of the analytic job. As we can see in Fig. 9, the time experience a linear increment in terms of the size of datasets. Note that, it takes linear time to execute *word_pair* job.

The results of processing roll-up queries were given in Fig. 10. Both the time cost for MongoDB and Redis demonstrates an sharper increment in terms of the size of the

Table 1 Average end-to-end query processing time with and without precomputing

Query Type	Without precomputing (second)	With precomputing (second)
FIND (sentiment_score = negative) where time = 07/07/17	4	3
FIND (sentiment_score = negative) AND (location near 153.3,-27) where time = 07/07/17	32	7
FIND (sentiment_score = negative) AND (location near 153.3,-27) AND (topic = food AND shop) where time = 07/07/17	140	9
FIND (sentiment score = negative) AND (location near 153.3,-27) AND (topic = food AND shop) where time > 07/07/17 AND time < 09/07/17	187	12
FIND (sentiment_score = negative) AND (location near 153.3,-27) AND (topic = food AND shop) where time = 07/07/17 OR time = 09/07/17	195	11
FIND (sentiment score = negative) AND (location near 153.3,-27) AND (topic = food AND shop) where time > 07/07/17 AND time < 09/07/17 OR time > 08/08/17 AND time < 10/10/17	220	24

datasets. This is because of the merging process is similar to the table-join of the relational database whose time consumption may grow quickly when the data size gets bigger. However, through a proper tune of the index interval, we can achieve a reasonable response time in practice. The MongoDB shows a slightly better performance than Redis, this shares the same reason when answering selectivity query. It takes more time for Redis to assemble the precomputed results for a given indexed collection.

4.3 Practical scenario

When the choice of an optimal execution plan is not critical, MongoDB is shown to be a viable alternative compared to relational databases [16]. We implement the precomputing system in a real-world social media analysis project, the HumanSensor. We use Twitter dataset as the main source for analytic tasks. Each tweet size is approximately 3 kB which is stored in MongoDB collection with each collection indexed in a per-day timestamp. Tweets carry some information such as text, GPS location, and the time posted. Based on these attributes, we define some sub-tasks to monitor public's opinion, urban activities, and topic modelling with the time interval as the parameter. Figure 11 represents the main idea of the system where the first query is divided into sub-tasks which will be queried according to temporal information.

Opinion mining is measured through the sentiment analysis which calculates the "good and bad" based on the sentiment score. The process itself relies purely on the tweet (text) which is segmented based on supervised learning. *Urban activities* on the other hand is determined by the GPS location provided in the tweet attributes (see Fig. 2). We visualise each location point as the heatmap to determine the density of urban movements. By visualising the location, we can predict the traffic congestion towards a certain period of time. Lastly, the *topic modelling* is used to capture the trending topic of people's interest in a certain period. By combining these three aspects we can easily understand the general point of view of a certain region.

The main problem of this complex big data analytic is the processing efficiency of collecting raw data into the final front-end visualisation. Normally, the process will go through several steps such as data cleaning, segmentation, clustering, and score weighting. There are two ways of processing data in a typical real world application, first is to directly grab raw data and then process them in the application layer (front-end). Second is by processing data in the database (back-end) and then send the result to the application. Both ways are acceptable, however, when data size is increasing, the application processing becomes a major bottleneck. As an example, in our use case we use NodeJS[6] as the front-end application where Fig. 12 depicts the two processes described above. The dotted line indicates the raw data are processed in the application layer.

Finally, we measure the end-to-end metrics of information retrieval starting from user query to the sub-tasks processing within the database in real-time. Table 1 shows the average processing time of the end-to-end platforms for various NoSQL query-based. We measure the query on a total of one million tweets. Compared to the traditional processing without the precomputation, the overall time is reduced from minutes to seconds. When the query complexity increase, the processing time grows exponentially due to additional MapReduce processing task which is further affected by network bandwidth cost. On the other hand, the precomputing depends on the number of collections since it only has to *scan* collections that fall into the query. With the precomputing architecture, we are able to speed-up the temporal merging which enables the real-time processing for complex task over big data.

[6] Node.js, https://nodejs.org/en/.

5 Conclusion

We presented a precomputing architecture suitable for NoSQL databases in particular MongoDB and Redis to answer temporal analytic queries. Within the architecture, we proposed indexing techniques, results storage structures as well as query processing strategies. Based on proposed architecture we are able to efficiently answer *drill-down* and *roll-up* temporal queries over large amount of data with fast response time. Through integration in real project running in Big data and Smart Analytics lab at Griffith University and experimental performance study we proved the effectiveness of our architecture and demonstrated its efficiency under different settings. We also showed that document-based store can outperform key-value store when the data fit in the memory. Considering future works it would be interesting to consider include extending the precomputed results over spatial data and to enable distributed join for merging functions, which will enable parallel join and hopefully reduce time threshold per collection.

Acknowledgements This project was partly funded through a National Environment Science Program (NESP) fund, within the Tropical Water Quality Hub (Project No: 2.3.2).

References

1. Becken, S., Stantic, B., Alaei, A.R.: Sentiment analysis in tourism: capitalising on big data. J. Travel Res. p. 0047287517747753 (2017). 10.1177/0047287517747753
2. Burdick, D., Doan, A., Ramakrishnan, R., Vaithyanathan, S.: Olap over imprecise data with domain constraints. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 39–50. VLDB Endowment (2007)
3. Chaudhuri, S., Dayal, U.: An overview of data warehousing and olap technology. ACM Sigmod Rec. **26**(1), 65–74 (1997)
4. Chen, C., Yan, X., Zhu, F., Han, J., Philip, S.Y.: Graph olap: towards online analytical processing on graphs. In: Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on, pp. 103–112. IEEE (2008)
5. Chevalier, M., El Malki, M., Kopliku, A., Teste, O., Tournier, R.: Implementing multidimensional data warehouses into nosql. In: International Conference on Enterprise Information Systems (ICEIS 2015), pp. 172–183 (2015)
6. Codd, E.F., Codd, S.B., Salley, C.T.: Providing olap (on-line analytical processing) to user-analysts: an IT mandate. Codd and Date **32** (1993)
7. Coronel, C., Morris, S.: Database systems: design, implementation, and management. Cengage Learn. (2016)
8. Cuzzocrea, A., Bellatreche, L., Song, I.Y.: Data warehousing and olap over big data: current challenges and future research directions. In: Proceedings of the Sixteenth International Workshop on Data Warehousing and OLAP, pp. 67–70. ACM (2013)
9. Dede, E., Govindaraju, M., Gunter, D., Canon, R.S., Ramakrishnan, L.: Performance evaluation of a mongodb and hadoop platform for scientific data analysis. In: Proceedings of the 4th ACM Workshop on Scientific Cloud Computing, pp. 13–20. ACM (2013)
10. Doulkeridis, C., Nørvåg, K.: A survey of large-scale analytical query processing in mapreduce. VLDB J. **23**(3), 355–380 (2014)
11. Franciscus, N., Ren, X., Stantic, B.: Answering temporal analytic queries over big data based on precomputing architecture. In: Intelligent Information and Database Systems—9th Asian Conference, ACIIDS 2017, Kanazawa, Japan, April 3–5, 2017, Proceedings, Part I, pp. 281–290 (2017)
12. Gudivada, V.N., Rao, D., Raghavan, V.V.: Renaissance in database management: navigating the landscape of candidate systems. IEEE Comput. **49**(4), 31–42 (2016)
13. Inmon, W.H., Hackathorn, R.D.: Using the Data Warehouse, vol. 2. Wiley, New York (1994)
14. Lin, C.X., Ding, B., Han, J., Zhu, F., Zhao, B.: Text cube: Computing ir measures for multidimensional text database analysis. In: Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on, pp. 905–910. IEEE (2008)
15. Lo, E., Kao, B., Ho, W.S., Lee, S.D., Chui, C.K., Cheung, D.W.: Olap on sequence data. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 649–660. ACM (2008)
16. Mahmood, K., Risch, T., Zhu, M.: Utilizing a nosql data store for scalable log analysis. In: Proceedings of the 19th International Database Engineering & Applications Symposium, pp. 49–55. ACM (2015)
17. Ntarmos, N., Patlakas, I., Triantafillou, P.: Rank join queries in nosql databases. Proc. VLDB Endow. **7**(7), 493–504 (2014)
18. Qi, Y., Candan, K.S., Tatemura, J., Chen, S., Liao, F.: Supporting OLAP operations over imperfectly integrated taxonomies. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 875–888. ACM (2008)
19. Scriney, M., Roantree, M.: Efficient cube construction for smart city data. In: Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (2016)
20. Stantic, B., Pokorny, J.: Opportunities in big data management and processing. Front. Artif. Intell. Appl. **270**, 15–26 (2014). (IOS Press)
21. Stonebraker, M.: SQL databases v. NoSQL databases. Commun. ACM **53**(4), 10–11 (2010)
22. Tao, Y., Papadias, D.: The mv3r-tree: a spatio-temporal access method for timestamp and interval queries. In: Proceedings of Very Large Data Bases Conference (VLDB), 11-14 September, Rome, pp. 431–440 (2001)
23. Wu, L., Sumbaly, R., Riccomini, C., Koo, G., Kim, H.J., Kreps, J., Shah, S.: Avatara: OLAP for web-scale analytics products. Proc. VLDB Endow. **5**(12), 1874–1877 (2012)
24. Zhao, H., Ye, X.: A practice of TPC-DS multidimensional implementation on NoSQL database systems. In: Technology Conference on Performance Evaluation and Benchmarking, pp. 93–108. Springer (2013)
25. Zhao, P., Li, X., Xin, D., Han, J.: Graph cube: on warehousing and OLAP multidimensional networks. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pp. 853–864. ACM (2011)