Answering Temporal Analytic Queries Over Big Data Based on Precomputing Architecture

Nigel Franciscus, Xuguang Ren and Bela Stantic

Institute for Integrated and Intelligent Systems, QLD, Australia
{n.franciscus, x.ren, b.stantic}@griffith.edu.au

Abstract. Big data explosion 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 (OLAP), 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 few works are studying the precomputing structure deliberately within 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. By using the precomputing structure, we are able to answer the *drill-down* and *roll-up* temporal queries over large amount of data within reasonable response time.

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 again 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 grow by 64% per year according to a recent report [1]. 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 its complexity poses big challenges in the processing of these queries. Aiming to tackle these challenges and to enhance the performance of analytic query processing, the concept of data warehouse [8] and OLAP [4] have been introduced since 1960s and many further works around these concepts have been proposed later on.

Data warehouses integrate data from different data sources into large repositories. The data within data warehouses are normally structured as denormalise multidimensional cubes to reduce the cost of heavy table joins. A large part of

modern OLAP systems are built on top of data warehouses which are stored with additional information. 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 precomputed results, conduct proper merging tasks and then quickly return 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 database is struggling in handling large amount of data [10]. The rise of NoSQL [11] databases 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 collected into NoSQL databases. Consequently, it has become an urgent demand to process analytic queries based on NoSQL databases efficiently. Some recent works are extending the techniques of data warehouses and OLAP into NoSQL. The work of [9] present strategies for constructing cubes from NoSQL stores. In contrast, the work in [3] gives the rules in converting existing cubes into NoSQL stores. However, there are few works focusing on the precomputing structure deliberately for NoSQL databases.

Motivated by the above issue, in this paper we present an architecture for answering temporal analytic queries over big data. As the time is an essential dimension for most of data analytic platforms, we choose temporal queries aspect as our focus in this paper and as the starting point of our work. We may extend our work into other dimensions in future works. The basic idea of our architecture is to divide the original NoSQL 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.

Contribution In this paper, we present a precomputing architecture for answering analytic queries for NoSQL data stores. Based on our architecture, we are able to answer the *drill-down* and *roll-up* temporal queries over large amount of data within fast response time. To be specific,

(1) We proposed the technique to index raw data into separated and smaller chunks based on temporal interval.
(2) We designed the storage structures for the precomputed results within MongoDB and Redis.
(3) We design three types of query models along with the strategies to answer each query type.
(4) We conducted extensive experiments to demonstrate the performance of answering three types of queries based on our architecture.

Organization The rest of the paper is organised as follows: in section 2, we give some related works; in section 3, we present the details of our precomputing

architecture; in section 4, we provide the experiment results; and finally in section 5 we conclude the paper and indicate some future work.

2 Related Work

In this section, we present some related works which are into classed into two categories.

(1) *NoSQL Database* According to a survey in [7], there are more than 100 No-SQL 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 our work, we used two NoSQL databases, MongoDB and Redis. The pros and cons are not our focus as MongoDB and Redis 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 de-normalised multidimensional table in relational database [2][5]. A large part of modern OLAP systems are built on top of data warehouses and utilize the cubes when processing analytic queries [6]. The work presented in [12] focuses on the time-range queries on relational cubes. There are some recent works extending the techniques of data warehouses and OLAP into NoSQL. The work of [9] present strategies for constructing cubes from NoSQL stores. In contrast, the work in [3] gives the rules in converting existing cubes into NoSQL stores. However, only few works studying the precomputing structure deliberately within NoSQL databases.

Contrast to previous work, we focus on the processing of analytic queries for NoSQL databases where no data is stored in relational database. We propose an index structure based on which we can answer the *drill-down* and *roll-up* queries over large amount of data within fast response time. Similar to work in [12], we particularly focus on the temporal queries with the time-range as the query parameter.

3 Precomputing Architecture

In this section, we present our detailed design of the precomputing architecture. We first give an overview of the architecture. Then we study the components respectively, which are: *(1) Raw Data Indexing*, *(2) Precomputed Results Structure* and *(3) Query Answering*.

3.1 Overview

In this subsection, we give an overview of our precomputing architecture as shown in Figure 1. It can be divided into several inter-related components: (i) *Raw Data Indexing*, where we collect the raw Twitter data and store them into NoSQL database(MongoDB) as time-indexed collections. (ii) *Precompute Results Structure*, where we execute analytic jobs (MapReduce based on Hadoop) and then store the precomputed results into NoSQL database (MongoDB and Redis). (iii) *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 by 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.

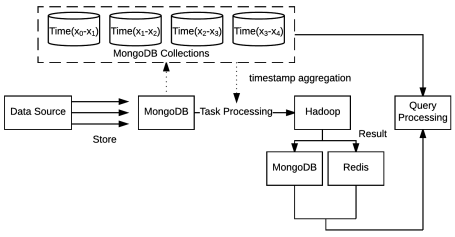

Fig. 1: Pre-computing Architecture

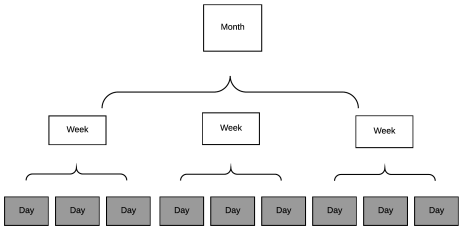

Fig. 2: Time Interval Index Structure

We present more details about each chosen specific ingredient.

Data Source As shown in Figure 1, we use Twitter as the data source in our 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 [13]. The tweets are formatted into JSON and they include the textual content as well as the posted time.

Database Platform The Twitter data is in JSON format which is naturally supported by the MongoDB. We choose MongoDB to store the raw data in our case study. MongoDB provides some features from relational databases like sorting, compound indexing and range/equal queries. Additionally, MongoDB has its own aggregation capability and in-house MapReduce operation.

Analytic Jobs Computing word frequency is a widely used analytic job in many literatures. Its intuitive application is the word cloud 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. Although MongoDB ships with in-house MapReduce, it has poor analytic libraries compared to Hadoop. We store the computed results into both Redis and MongoDB.

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 [12]. 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, we group tweets 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 Figure 2, we choose a day as the time interval. The tweets posted on the same day (grey box) will be grouped into the same collection. It is worth noting that a week is a super-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 Figure 2. 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 one day, we first lock down to the tweets collections within this range and load their corresponding precomputed results. Then we merge these results together and get the final results for 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*.

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 Figure 3, the name of the result collection is *1475118067000*(29 Sep 2016). The *hello* and *world* co-occurred in 100 tweets which are posted on the day of 29 Sep 2016.

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. 3: MongoDB Result Structure

Key	Value
1475118067000_hello_world	100
1475118067000_happy_world	70
1474315055000_hello_world	60

Fig. 4: Redis Result Structure

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 when given a specific timestamp and/or word-pair. As we can see in the example in Figure 4, 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).

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: (i) Single

Selectivity Query, (ii) Drill-down Query, (iii) Roll-up Query as we can see in the following models:

(i) Single Selectivity Query
QUERY data WHERE $time = T_x$ WITH $Gra(time, T_x) = \Phi$.

(ii) Drill-down Query
QUERY data WHERE $time > T_x$ AND $time < T_y$ AND T_y-$T_x < \Phi$ WITH $Gra(time, T_x, T_y) < \Phi$.

(iii) Roll-up Query
QUERY data WHERE $time > T_x$ AND $time < T_y$ WITH $Gra(time, T_x, T_y) = \Phi$.

In the above models, we use Φ to denote the interval when we index the raw data (as mentioned in Subsection 3.2). The function $Gra(t)$ is to decide the granularity of the time parameter t, for example, $Gra(12\text{am } 15 \text{ Sep } 2016) = hour$ and $Gra(15 \text{ Sep } 2016) = day$. Intuitively, *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. Consider the following example where each one query corresponds to one query type respectively.

(i) Word-pair frequency on 02/April/2016.

(ii) Word-pair frequency from 9:00pm of 08/April/2016 to 11:00pm of 08/April/2016.

(iii) 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. We present a basic algorithm to merge multiple result MongoDB collections here, as shown in Algorithm 1.

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

Algorithm 1: MergeResults

Input: Multiple precomputed results $T = \{T_1 \dots T_n\}$
Output: final result R

```
1  HashMap H ← ∅
2  for each collection T_k ∈ T do
3      for each document w ∈ T_k do
4          if w.word_1_w.word_2 is not in H then
5              H(w.word_1_w.word_2) ← w.frequency
6          end
7          else
8              H(w.word_1_w.word_2) ← H(w.word_1_w.word_2) + w.frequency
9          end
10     end
11 end
12 Result R ← JSON(H)
13 return R
```

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 1, we omit it here.

It is worth noting that a larger interval leads to a larger index 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[1]. In this section, we present our experiment results so as to study the response time of the query answering under different data settings.

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 index the 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.

[1] HumanSensor project is a data mining application conducted by Griffith University and sponsored by Gold Coast City Council

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 @ 3.10GHz with 4GB RAM. We used Hadoop (2.6.0), MongoDB (3.2.9) and Redis (3.0.1).

4.2 Results and Analysis

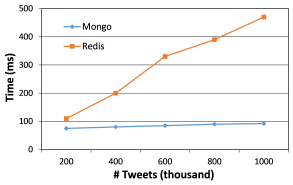

Fig. 5: Single Selectivity

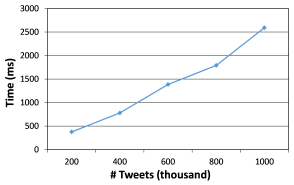

Fig. 6: Drill-down

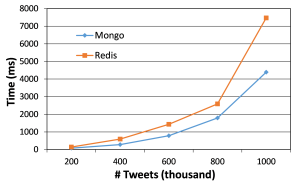

Fig. 7: Roll-up

The results of processing single selectivity query are given in Figure 5. 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 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 6 presents the results of answering drill-down query. As discussed in Section 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 Figure 6, 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 Figure 7. Both the time cost for MongoDB and Redis demonstrates an sharper increment in terms of the size of the 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.

5 Conclusion

We presented a precomputing architecture based on NoSQL database to answer temporal analytic queries. Within the architecture, we propose the indexing techniques, results storage structures and the query processing strategies. Based on our architecture, we are able to answer the *drill-down* and *roll-up* temporal queries over large amount of data within fast response time. Through integration in practical project and the performance study in our experiments, we proved the effectiveness of our architecture and demonstrated its efficiency under different settings. Some future works include extending the precomputed result over spatial data and to enable the distribute join for the merging function. That way, we can do parallel join to reduce the time threshold per collection.

References

1. Knowledge Management, http://www.globalgraphics.com/technology/ knowledge-management/
2. Chaudhuri, S., Dayal, U.: An overview of data warehousing and olap technology. ACM Sigmod record 26(1), 65–74 (1997)
3. 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)
4. 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)
5. Coronel, C., Morris, S.: Database systems: design, implementation, & management. Cengage Learning (2016)
6. 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)
7. Gudivada, V., Rao, D., Raghavan, V.: Renaissance in data management systems: Sql, nosql, and newsql. Computer (2015)
8. Inmon, W.H., Hackathorn, R.: Using the data warehouse. 1994. New York ua: John Wiley & Sons, Inc (1994)
9. Scriney, M., Roantree, M.: Efficient cube construction for smart city data. In: Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (2016)
10. Stantic, B., Pokornỳ, J.: Opportunities in big data management and processing. Frontiers in Artificial Intelligence and Applications, vol. 270. IOS Press (2014)
11. Stonebraker, M.: Sql databases v. nosql databases. Commun. ACM 53(4), 10–11 (Apr 2010)
12. 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 (2001)
13. Yu, Y., Wang, X.: World cup 2014 in the twitter world: A big data analysis of sentiments in us sports fans tweets. Computers in Human Behavior 48, 392–400 (2015)