Maxson: Reduce Duplicate Parsing Overhead on Raw Data

Xuanhua Shi[†], Yipeng Zhang[†], Hong Huang[†*], Zhenyu Hu[†], Hai Jin[†], Huan Shen[†], Yongluan Zhou[‡], Bingsheng He[§], Ruibo Li[¶], Keyong Zhou[¶]
[†]National Engineering Research Center for Big Data Technology and System/ Services
Computing Technology and System Lab, Huazhong University of Science and Technology, China
[‡]Department of Computer Science, University of Copenhagen, Denmark
[§]National University of Singapore
[¶]Alibaba Group, China
[†]{xhshi, jinyu_zyp, honghuang, cszhenyuhu, hjin, huanshen}@hust.edu.cn,
[‡]zhou@di.ku.dk, [§]hebs@comp.nus.edu.sg, [¶]{ruibo.lirb, zky.zhoukeyong}@alibaba-inc.com

***Abstract*—JSON is a very popular data format in many applications in Web and enterprise. Recently, many data analytical systems support the loading and querying JSON data. However, JSON parsing can be costly, which dominates the execution time of querying JSON data. Many previous studies focus on building efficient parsers to reduce this parsing cost, and little work has been done on how to reduce the occurrences of parsing. In this paper, we start with a study with a real production workload in Alibaba, which consists of over 3 million queries on JSON. Our study reveals significant *temporal* and *spatial* correlations among those queries, which result in massive redundant parsing operations among queries. Instead of repetitively parsing the JSON data, we propose to develop a cache system named Maxson for caching the JSON query results (the values evaluated from JSONPath) for reuse. Specifically, we develop effective machine learning-based predictor with combining LSTM (*long short-term memory*) and CRF (*conditional random field*) to determine the JSONPaths to cache given the space budget. We have implemented Maxson on top of SparkSQL. We experimentally evaluate Maxson and show that 1) Maxson is able to eliminate the most of duplicate JSON parsing overhead, 2) Maxson improves end-to-end workload performance by 1.5–6.5×.**

***Index Terms*—JSON parsing, semi-structured format, data analytics system.**

I. Introduction

In many Web and enterprise applications, JSON is a highly popular data exchange format. The attractive features of JSON include flexibility, simplicity, human-readability, and high expressive power. As such, nine of the ten most popular Web APIs (mainly composed of APIs provided by big companies such as Google, Facebook, and Twitter) expose data in JSON format [1]. Many data analytics engines (e.g. Spark [2], Flink [3], Storm [4], Drill [5]) natively support loading and querying JSON data. However, unlike relational data, JSON is a semi-structured format and thus has to be parsed before further analysis and query. Recent research indicates that a key bottleneck in querying raw data is parsing the data itself, about 80% execution time spent on parsing JSON data [6].

*Corresponding author

Many previous studies focus on building efficient parsers to reduce this parsing cost [6]–[9]. Mison [6] utilizes the SIMD instruction to build a *field index* for a JSON string based on a special structure character such as brackets and colons. Sparser [7] develops a new approach that apply filters on raw byte stream before parsing based on the observation that many real-world applications have high selectivity. Despite the significant improvement in parsing speed, the overhead can still be large for a large amount of JSON and complex JSONPath. In reality, many enterprises face the challenging problem of executing a large number of queries on a large amount of JSON data every day across thousands of servers. Thus, besides improving the parsing speed, it is also important to research on how to reduce the occurrences of parsing.

In this paper, we start with a study with a real production workload from Alibaba. In Alibaba, JSON format is widely used to record information generated from various business and analytics tasks, and the data volume continues to grow rapidly. The workload in our study is a five-month trace which consists of about 3 million queries on JSON. The study reveals that high repeatability of JSONPath executions in those queries. We find that over 89% of JSON parsing traffic is spent on repetitive JSONPath executions across different queries. Since each query needs to parse JSON once, massive parsing redundancies cause significant waste in cluster resources.

Motivated by the previous study on big data workloads [10], we find that the JSONPath redundancy is due to significant temporal and spatial correlations among queries.

- *Temporal correlations:* The queries exhibit temporal correlations where it is common to have a series of queries involving the same JSON parsing on the same data set in different time windows. Note that new JSON data are stored in the production cluster periodically. For example, one data set might store sale logs and is appended daily with new entries. A query might be issued daily to retrieve the most sales items from the sale logs over 3-day sliding windows. These daily queries have clear temporal correlations.

- *Spatial correlations:* The queries also exhibit spatial correlations where a data set is often the target of multiple queries involving overlapping or even the same JSONPath parsing. Take the same sale logs as an example. A query might probe the most turnover items in the same 3-day window, thereby exhibiting spatial correlations with the first set of queries and both queries will parse *item_name* and *item_id* (see in Fig. 1).

Based on the above observations, we propose to develop a cache system named Maxson[1] for caching the JSON query results (the values evaluated from JSONPath) for reuse. Maxson leverages high JSONPaths repeatability in the workload to avoid duplicate parsing overhead across different queries. There have been many previous studies on improving caching systems under different contexts [11]–[14]. They usually have rather sophisticated caching policies and advanced replacement policies. In our study, we look for a simple and sufficiently effective caching strategy for the production workload. Also, we hope that our system can cause minimal changes to existing data analytics platforms. Specifically, Maxson features the following two simple and efficient designs.

First, we develop effective machine learning-based predictor with combining LSTM (*long short-term memory*) and CRF (*conditional random field*) to determine the JSONPaths to cache given the space budget. We call the JSONPath with the parsed times greater than or equal to twice in one day as ***Multiple-Parsed JSONPath (MPJP)***. *MPJP* are candidates for caching with benefits. Every midnight, we predict and pick the MPJP which can minimize the execution under limited storage resources. After that we parse their values from the raw data table and cache them into a new cache table with the same storage format as the raw data table. Those parsing operations can be done at the non-peak hours of the cluster. Then we can read the value directly from the cache table without parsing overhead for JSONPath cache hits.

Second, to enable queries to make use of the cached JSONPaths, a query plan modifier is implemented. Because of the cache, a part of the data of each record comes from the raw data table, and other part of the data comes from the cache table. These two parts of data belonging to the same row need to be stitched together correctly to form a complete record. The naive method is to join the raw data table and cache table to find the complete record, but the join operations can be costly. To do alignment efficiently, we initialize two parallel and synchronized readers: one is responsible for reading the cache table and the other is responsible for reading the raw data table. We design synchronized methods to guarantee that the two readers will read the data from the same row. Furthermore, for queries with filtering predicates on the values of cached JSONPaths, we also implemented predicate pushdown onto the cache table to maximize query performance.

[1]The source code is available at https://github.com/CGCL-codes/Maxson

II. Preliminaries

A. JSON Data in Data Warehouses

Large-scale data warehouses (eg. Hive [15], MaxCompute [16]) typically store a large volume of data in distributed data storage and provide SQL-Like language to manage and query data. In these systems, JSON data is often stored as *String Types*. Before running queries on such data, it is necessary to parse the JSON data, to extract and transform them into the native data format supported by the system. Fig. 1 shows an example of how JSON data are typically stored in a data warehouse. Table *T* in database *mydb* has three columns: *mall_id*, *date*, and *sale_logs*, which store sales information in JSON format. Two example queries retrieve the items with the highest turnover and the highest sale count in a 3-day window, respectively. In these two queries, the function *get_json_object* is used to parse the JSON string and retrieve the specified fields such as *turnover*, *sale_count*. The function *get_json_object* requires two parameters: namely, *Column Name*, which specifies which column to read the JSON string, and JSONPath, which indicates the path to the specified field in a JSON string. For example, the JSONPath *$.turnover* indicate the path to field *turnover* in *sale_logs*. In summary, to read the value of a field, one has to specify the *database name*, the *table name*, the *column name*, and the JSONPath.

Information table **T** in database **mydb**

mall_id	date	sale_logs
0001	20190101	{"item_id" : 000001, "item_name" : "apple", "sale_count" : 10, "turnover":20, "price": 2 ...}
0001	20190102	{"item_id" : 000002, "item_name" : "watermelon", "sale_count" : 5, "turnover":50, "price": 10 ...}
.....		
0001	20190131	{"item_id" : 000003, "item_name" : "banana", "sale_count" : 30, "turnover":90, "price": 3 ...}

select mall_id, get_json_object(sale_logs, '$.item_id') **as** item_id, get_json_object(sale_logs, '$.item_name') **as** item_name, get_json_object(sale_logs, '$.turnover') **as** turnover **from** mydb.T **where** date between '20190101' and '20190103' order by get_json_object(sale_logs, '$.turnover') **limit 1**

select mall_id, get_json_object(sale_logs, '$.item_id') **as** item_id, get_json_object(sale_logs, '$.item_name') **as** item_name, get_json_object(sale_logs, '$.sale_count') **as** sale_count **from** mydb.T **where** date between '20190101' and '20190103' order by get_json_object(sale_logs, '$.sale_count') **limit 1**

Fig. 1: Query data in JSON format

B. Data Update Pattern

We further study the time of table updates during the day. As shown in Fig. 2, updates are more frequent at noon, but rare at midnight. In addition, these updates usually come from the data generated in the previous day. Furthermore, the data that has been appended will hardly be changed, only 2% of the tables experienced modification of previously appended data.

C. Cost of Parsing JSON Data

In order to demonstrate the overhead of parsing JSON data, we tested three types of queries, which appear very frequently in the collected production workload from Alibaba. Q1 is a simple **SELECT** query which retrieves two attributes from the queried JSON data. Q2 uses **COUNT** aggregate function with a group-by condition. Q3 performs a self-equijoin. We run the queries on the JSON data generated from Nobench

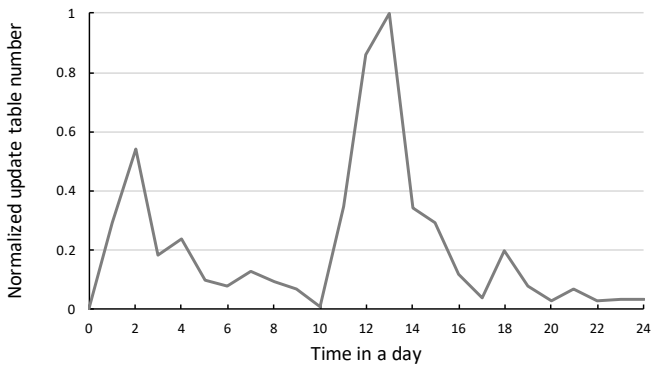


Fig. 2: Time of table updates during the day

[17]. Fig. 3 show the composition of the running time of these three queries running on JSON data using SparkSQL. It is very obvious that parsing JSON data accounts for the majority of the execution time ($\geq 80\%$), even both simple data retrieval (Q1) and more expensive queries (Q2 and Q3) involving joins and aggregations.

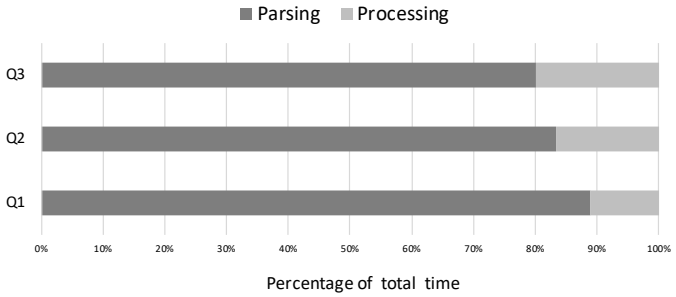


Fig. 3: Parsing and query processing cost in three common types of query

D. Workload Analysis

In Alibaba, there are about 6,000 machines allocated to the service of querying JSON data. The queries are mainly data mining tasks on JSON data generated from a wide range of services inside Alibaba such as sale logs, machine state logs. The data generated by these services are typically loaded into the data warehouse on a daily basis. We collected a 5-month query trace from a deployed data analysis system with over several thousands of machines. We retrieved the information related to the trace, which includes the query strings, table update timestamps, query submission timestamps, query plans. The trace contains nearly 3 million of successfully executed queries. These queries are run on around $24,000$ tables stored in a reliable append-only distributed file system similar to HDFS [18]. We conducted the following analysis of the queries in the trace.

1) Temporal correlation: Queries in the trace exhibit strong temporal correlations: 82% queries are recurring, which are submitted by about $1,900$ users. Out of these queries, over 71% are repeated daily (usually with a calendar day window, e.g., requiring the data from yesterday, while approximately 7% with a window spanning multiple days); 17% are repeated weekly, mostly with a window across a week.

2) Spatial correlations: We found that the frequency of JSONPath parsing closely follows the power-law distribution: 89% of the parsing traffic are on 27% JSONPaths. This distribution reveals spatial correlations across different queries. It is often the case that users analyzed the same data in different dimensions in different queries. For example, the aforementioned queries shown in Fig. 1 are daily queries extracted from the Alibaba workload, which analyze, respectively, the turnover and sale count values from the same input data. While these two queries parse different JSONPaths, they also parse common JSONPath from the sale logs, such as $item\_name$, $item\_id$. Fig. 4 depicts the number of queries that involve each particular JSONPath. On average each JSONPath will be requested by 14 queries.

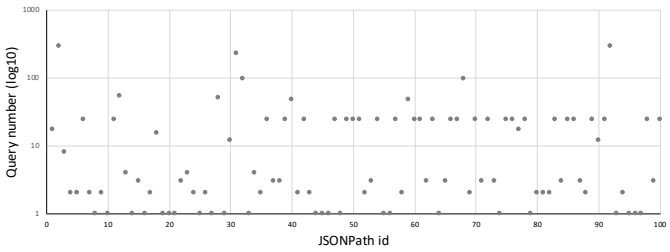


Fig. 4: Number of queries that contain each JSONPath. Each JSONPath is given a unique id, and the numbers of queries that contain these JSONPaths are reported.

Finally, queries usually only involve data that are loaded before the previous day. In other words, queries would not involve data loaded on the same day, which provides time for checking and validating data after they are loaded.

III. System Design

A. Technical Challenges

The above study of the trace reveals strong temporal and spatial correlations across queries, which lead to significant JSON parsing redundancies. Inspired by the previous work such as [11]–[14], we propose a cache-based approach to reduce the parsing overhead. Frequently accessed JSON data is parsed and stored in the cache so that query execution do not need repeatedly pay the parsing overhead.

However, conventional online caching methods where data are first cached when it is accessed and cache is maintained using an online replacement strategy, such as LRU, would not be a desirable choice in our scenario. In our scenario, the data accessed by queries is updated daily, and once updated, it is usually remained unchanged within a day. Furthermore, queries would only involve data that are loaded yesterday or earlier. Therefore, there exist ample opportunities to make use of unutilized cluster resources to parse the JSON data and pre-load the cache even before the data are accessed the first time. This can often happen during midnight when the cluster

is under-utilized. Finally, with online caching methods, the first queries that access a JSONPath can not take advantage of the cache, therefore it does not provide a uniform and fair user experience.

B. System Overview

Our objective is to implement a lightweight JSONPath caching system that can reduce the execution time of user queries under the constraint of storage resource. To avoid the aforementioned problems of online caching and replacement methods, Maxson adopts a prediction-based caching approach, where the system predicts daily the JSONPaths that would be accessed in the coming day, and it pre-parses the selected JSONPaths and stores the data values into the cache.

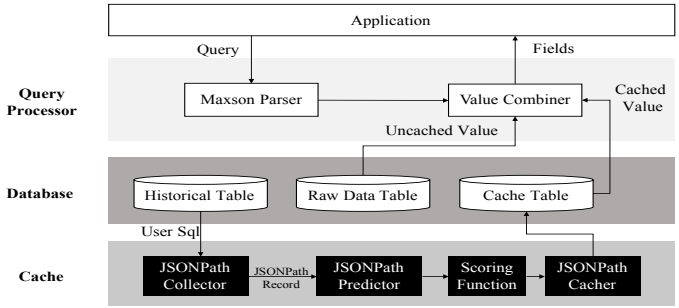


Fig. 5: System architecture of Maxson

Fig. 5 depicts Maxson's overall architecture. *JSONPath Collector* collects historical user queries, and for each JSONpath, extracts information about its location (i.e. *database name*, *table name*, *column name*) and the number of times that it is accessed. Such information is stored in a statistics table, which is partitioned by date.

The collected statistics are used to train the models in *JSONPath Predictor*, which is used to predict the candidate JSONPaths to be cached for the coming day, which are called **Multiple-Parsed JSONPaths (MPJP)**. Then a *Scoring Function* would be used to choose the MPJPs that can provide the highest benefit in terms of minimizing query execution time. Then *JSONPath Cacher* parses the values of the chosen *MPJPs* into a cache table at midnight when updates rarely happen. The cache is emptied re-populated every midnight. The storage size used by the cache table is a parameter that can be tuned according to the available resource.

Maxson is designed to be fully compatible with SparkSQL, a Spark module for structured data processing. Users can execute SQL queries and support reading and writing data stored in Hive. SparkSQL compiles SQL queries into physical plans to be executed on a cluster. A physical plan is a set of RDD [2] operations that are executed on the data source, typically contains *scan*, *filter*, *projection*, *join*, etc. To make a physical plan to access the cache table, we implemented *MaxsonParser* based on SparkSQL parser. When the *MaxsonParser* compiles SQL statement into a physical plan, if a JSONPath in the SQL statement hits a valid cached value, it generates a placeholder that stores the JSONPath information and the reference to the cache table. A cache item is valid if the cached time is behind the last modification time of raw data table. If the query needs to access both cached and uncached data, then during the table *scan* phase, we use *Value Combiner* to stitch the cached and uncached data into complete records.

IV. Implementation

A. JSONPath Predictor

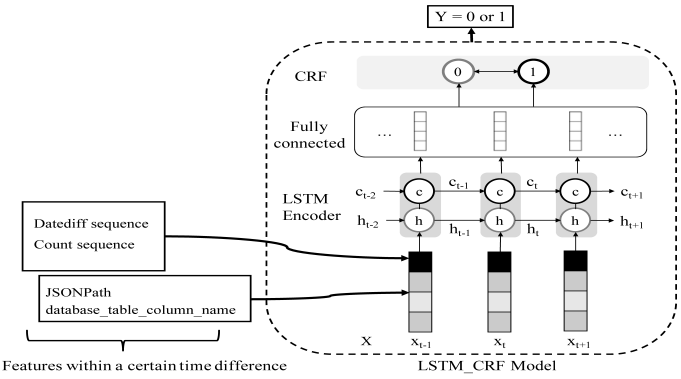


Fig. 6: Overview of JSONPath predictor

Fig. 6 depicts the overview of JSONPath Predictor. It predicts whether a given JSONPath will be accessed at least twice on the next day or not. In other words, it predicts if a JSONPath is a **Multiple-Parsed JSONPath (MPJP)** on the next day.

The input of the predictor is provided by *JSONPath Collector* which includes *JSONPath*, *Date*, and *Count*, which is access times of the JSONPath on the specified date. Our hypothesis is that the next day's MPJPs can be predicted by the access times of JSONPaths in the history, and the nearer history has greater impact on the prediction than the further history.

For each JSONPath, the features include *database name*, *table name*, and *column name*, *Count sequence*, which is a sequence of JSONPath's access times on each day, and *Datediff sequence*, which is a sequence of integers indicating how old each access times in *Count sequence* is (e.g. 1 indicating it is one day old and so on). We take *database name*, *table name* and *column name* as part of the feature set because JSONPaths in the same data source often appears together in the queries.

We use a hybrid model composed by *long short-term memory* (*LSTM* [19]) and *conditional random field* (*CRF* [20]) to implement Predictor. LSTM [19] can perform sequence feature extraction well with its core component c (cell state) and h (hidden state), where c is used to save the long-term state and h is to save what the model has learnt. Besides capturing the relation of features for learning like general classification algorithms, CRF [20] can learn the context relation of labels, which makes the model prediction more accurate. Combining LSTM with CRF, we can well capture temporal and spatial correlations of JSON queries.

For the given records of JSONPath features $X = x_1, ..., x_n$ where x_i is an input vector composed of the ith *JSONPath* location information, *Datediff sequence*, and *Count sequence*. The JSONPath records are first feed into LSTM layer, it adopts Cross-Entropy function [21] to adjust the correctness of model. Then the label sequence output by LSTM, is fed into the CRF layer, it adopts the viterbi algorithm [22] to decode the sequence to get the label with the maximum probability. We have also tried the other models, eg. LR [23], SVM [24], MLPClassifer [25], and LSTM [19]. LSTM+CRF hybrid model performs the best according to our experiments. The results are presented in Section V-B.

B. Scoring Function

TABLE I: Notation used in scoring function

Symbol	Description
B_j	Average size of the value of *MPJP*$_j$
A_j	Acceleration per byte of *MPJP*$_j$
M_i	Number of *MPJP*s in $query_i$
N_i	Number of JSONPaths in $query_i$
R_j	Relevance of *MPJP*$_j$
P_j	Average parsing time of *MPJP*$_j$
O_j	Average number of occurrence of *MPJP*$_j$
$Score_j$	Score of *MPJP*$_j$

Due to the storage constraint, we may not be able to cache the values of all the *MPJP*. We propose a scoring function to rank the predicted *MPJP* for caching. Table I summarizes the notations that are used in the description below. The score of a MPJP takes into account the data size after parsing, time needed to spend on parsing, and the relevance of the *MPJP*. The details of these factors are described as follow:

Acceleration per byte. The size of *MPJP* directly affects the number of *MPJPs* that we can cache. When the disk size is fixed, we prefer to cache the one that can bring more gains on performance. We define **acceleration per byte** of an *MPJP*, denoted by A_j as the average query time that can be saved by each byte of its parsed values (Eq. 1). We calculate average size of an *MPJP*, denoted by B_j, by sampling each split from the table, and its average parsing cost, denoted by P_j, using the same parsing algorithm in the data analysis engine.

$$A_j = \frac{P_j}{B_j} \tag{1}$$

Relevance. If all the JSONPaths of a query are cached, it is unnecessary to read the JSON strings from the raw data table to parse the values of the uncached JSONPaths and stitch them with the cached values. Instead, we can perform the cache-only reading, which is cheaper in terms of both I/O and CPU cycles. Note that only MPJPs are candidates for caching. Therefore, the higher percentage of JSONPaths accessed by a query being MPJPs, the larger portion of JSONPaths accessed by the query can be cached to bring higher benefit in terms of reducing query time. Based on this observation, we define the relevance of *MPJP* (Eq. 2) as follows. Suppose *MPJP*$_j$ is accessed by n queries. We count the total number of *MPJPs* and JSONPaths involved in each of these n queries, respectively, denoted by M_i and N_i. The relevance of the jth *MPJP*, denoted by R_j, is defined in Eq. 2, which is actually the fraction of JSONPaths accessed by the corresponding queries being *MPJP*s. We prefer caching *MPJP*s with higher R_j, so that it is more likely the case that all JSONPaths of some queries are cached. We use the same queries as *JSONPath Predictor* to calculate the *relevance*.

$$R_j = \frac{\sum_{i=1}^{i=n} M_i}{\sum_{i=1}^{i=n} N_i} \tag{2}$$

Number of occurrence. We define the number of occurrence of *MPJP*$_j$, denoted by O_j, as the number of queries that would access *MPJP*$_j$. The higher the value of O_j, the more queries that can be accelerated by caching *MPJP*$_j$.

Score. To take all the aforementioned factors into account, the score of *MPJP*$_j$ is defined as follows:

$$Score_j = A_j \cdot R_j \cdot O_j \tag{3}$$

C. JSONPath Cacher

At the start of the cache population time, which is usually scheduled at midnight every day, *JSONPath Cacher* receives from *JSONPath Predicator* a list of *MPJP*s sorted in descending order of their scores. It then caches the *MPJPs* in the sorted order until it runs out space.

The pre-parsing and caching operations are done in a scalable way using Spark. As Fig. 7 shows, the raw data table is stored in *ORC* [26] format on Hive, which contains multiple files. In HDFS, a file can be divided into one or more blocks (note that a block cannot span multiple files). In Spark, one or more blocks can form an *input split*, which corresponds to one Spark partition. The number of blocks contained in each input split is a tunable parameter in Spark. When caching the table, we treat a file as an input split to guarantee that the value parsed from the raw data table file will only be written to one cache table file. And We cache the JSONPath from the same raw data table into the same cache table. In order to remember the mapping relationship between the cache table and the raw data table, we name a cache table according to the corresponding database name and raw data table name, and a cache field according to the corresponding column name and JSONPath.

In order to correctly find the file where the parsed value is located, we modified the Spark naming function when writing the cache table file, so that the cache table file and the raw data table file have the same sorting order during the process of reading the table and the two readers can get the correct file with the same index.

D. Maxson Parser

When a user submits a job to SparkSQL, in order to take advantage of the cached tables, we transparently modify the physical plan for the table reading phase. Algorithm 1 shows the procedure of modifications to the physical plan. The input of the algorithm contains two data structures: *ProjectList* and *Predicate*. *ProjectList* contains the expressions that appear in

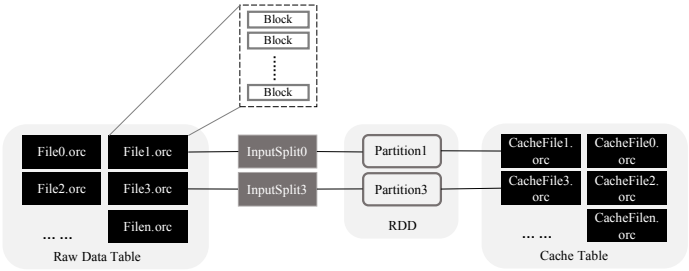


Fig. 7: Relationship between Spark Partition and HDFS files

Algorithm 1: Modifications to the Physical Plan

Input : ProjectList: PL, Predicate: P, DataBaseName: DBN, TableName: TN;
Output: ProjectList which has been replaced: $PL0$, Predicate which has been replaced: $P0$;

1 $PL0$ = `Replace`(ProjectList)
2 $P0$ = `Replace`(Predicate)
3 **Function** `Replace`(pl_{arg})
4 **foreach** *Expression expr in* ProjectList **do**
5 res = `MatchExpr`(pl_{arg});
6 **switch** res **do**
7 **case** $placeholder$ **do**
8 **return** $placeholder$
9 **otherwise do**
10 `Replace`($pl_{arg}.children$)
11 **Function** `MatchExpr`(e_{arg})
12 **begin**
13 **switch** e_{arg} **do**
14 **case** *get_json_object*(CN,JP) **do**
15 **if** (DBN,TN,CN,JP) *is cached* **then**
16 $cacheTime \leftarrow$ get the cache time of the cached table;
17 $modifyTime \leftarrow$ get the latest modification time of the raw data table;
18 **if** *the* $modifyTime$ *after the* $cacheTime$ **then**
19 mark the cache table invalid;
20 **return** *None* ;
21 **else**
22 $description \leftarrow$ ($CN.name$, $CN.id$, JP);
23 **return** *placeholder(description)*
24 **otherwise do**
25 **return** *None*

the **SELECT** statement and *Predicate* contains the expressions that appear in the **WHERE** statement. The columns referenced by these expressions determine the columns to be read at the table reading stage. Since expressions can contain subexpressions, we use a recursive function to iterate through

```
1  select
   non_json_column0,
3  non_json_column1,
   get_json_object(json_column0,'$.id') as
       json_column0_id,
5  get_json_object(json_column0,'$.url') as
       json_column1_url
   from T
7  where get_json_object(json_column0,'$.id') > 10000;
```

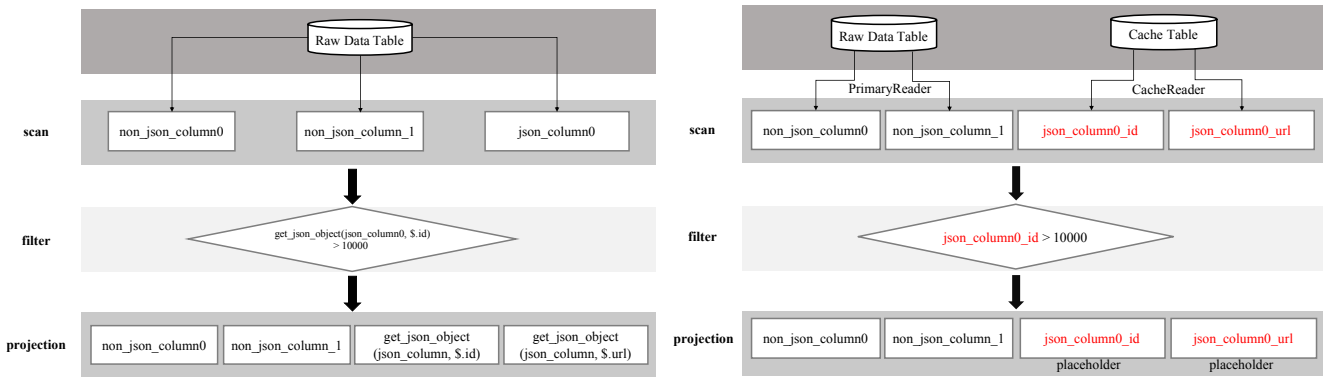
Fig. 8: An SQL example for our modified physical plan

each expression in the *ProjectList* and its subexpressions (lines 3-15). When an expression is the function *get_json_object*, we use pattern matching to extract the information of *CN* and *JP* from it. Based on *DBN* and *TN*, We determine whether we have cached this JSONPath (line 15). In lines 16-19, we get the last modification time of the table from its metadata (note that Spark loads metadata into memory before generating the physical plan), and compare with the time of the cache. If the cache time is before the last modification time, we mark the cache table as invalid. Invalid cache tables would be deleted when we perform caching operations next time. If the JSONPath access is a cache hit and the cache is valid, we replace *get_json_object* with a placeholder and put the *Column Name*, the id of the column expression and the JSONPath into the placeholder (lines 22-23). The information stored in the placeholder would be used by reader to read the cached values. On the other hand, if it is a cache miss, the expression would not be changed, and data would be read by following the normal procedure, i.e. reading the JSON string from the raw data table followed by calling the *get_json_object* function to parse its value.

We use an example to illustrate how this algorithm works. As shown in Fig. 8, it is a simple query that retrieve the values of *non_json_column0* and *non_json_column1* and parses the *id* and *url* in the *json_column0* from the table *T*. Fig. 9 shows the comparison between SparkSQL's original physical plan and the modified physical plan for this simple query. We assume that both *id* and *url* have been cached. The original physical plan need to project the three columns: *non_json_column0*, *non_json_column1*, *json_column0* and needs to pass *json_column0* to *get_json_object* to retrieve *id* and *url*. While in the modified plan, *json_column0* is removed, replaced by two placeholder named *json_column0_id* and *json_column0_url*. The values of *id* and *url* are read from the cache table instead of by parsing *json_column0*.

E. Value Combiner

A query may involve both cached values, such as *id* and *url* in the above example, and uncached values, such as *non_json_column0* and *non_json_column1*. We start two readers to read their values separately. The one that reads the uncached values called *PrimaryReader*, while the other one that reads the cached values is called *CacheReader*. We pass the column names that they need to read by the configuration of the readers. As Fig. 9 shows, *PrimaryReader* is responsible for reading *non_json_column0* and *non_json_column1* and



(a) Spark physical plan

(b) Maxson physical plan

Fig. 9: Comparison of physical plans generated by Spark and Maxson

CacheReader is responsible for reading *json_column0_id* and *json_column0_url*. These four columns need to be stitched together to form the complete records.

Algorithm 2 shows the procedure of *Value Combiner*. The input of the algorithm includes the result schema (*S*), the cached JSONPaths name (*J*), *PrimaryReader*, *CacheReader*, and the split index (*I*). The name of the column we output is recorded in the *schema*. In contrast to the original physical plan, the output schema has four columns instead of three, with the original *json_column0* being replaced by *url* and *id*. *JSONPath Cacher* guarantees that the two files with the same index have the same number of rows, and the cache table files and raw data table files are aligned. We treat a file as an input split when reading the table. The algorithm first reads the values of split *I* from raw data table and cache table (line 1-2). Then for each pair of *cache_value* and *uncached_value* that belong to the same row, we find their index from the schema according to their name and then put their values in the corresponding position of *V* (lines 5-8, 9-12). In particular, when one reader has no value to read, we will directly return the value of the other reader.

Algorithm 2: The Procedure of the Value Combiner

Input : output schema: S, cached JSONPath name: J, PrimaryReader: PR, CacheReader: CR, SplitIndex: I;
Output: The combined value V read from table;

```
1  uncachedSplit ← read split I from raw data table by PR;
2  cacheSplit ← read split I from cache table by CR;
3  foreach uncachedValues, cachedValues in uncachedSplit, cachedSplit do
4      if uncachedValues is empty then
5          return cachedVlues;
6      if cachedValues is empty then
7          return uncachedValues;
8      V = Value[S.length];
9      foreach cached_value, cached_name in cachedValues do
10         fieldIndex ← index of cached_name in schema S;
11         V[filedIndex] ← cached_value
12     foreach uncached_value, uncached_name in uncachedValues do
13         fieldIndex ← index of uncached_name in schema S;
14         V[filedIndex] ← uncached_value
15     return V;
```

F. Optimization with Predicate Pushdown

In our system, data is stored in the *ORC* [26] format, which is an efficient columnar storage format widely used by many analytical systems. An *ORC* file can be divided into multiple *stripe* depending on the data size. A *stripe* can contain multiple *row groups*, each containg a group of 10,000 rows. An *ORC* file maintains various indexes of the values within each column (e.g. min and max values). The indexes are used by the reader using Search ARGuments or SARGs, which are simplified expressions that can specify the rows that are of interest.

Since we cache JSONPath's values separately, if the predicate contains a cached JSONPath, we can use it to restrict the rows that should be read from the raw data table and cache table. Algorithm 3 shows the procedure of applying predicate pushdown. We use a query in Fig. 8 to illustrate how this algorithm works. As shown in Fig. 10, when we push the SARGs *id >10000* down to the cache table, the *CacheReader* initializes *row group* array from cache table's file with SARGs (line 4), and *row group0* that contains id less than 10000 is completely skipped. There is an array in *CacheReader* to record the skipped *row groups* [false, true,...] (line 5), where *false* and *true* indicate skipped and non-skipped respectively. We share the array to the *PrimaryReader* (line 7), so that the *PrimaryReader* would also skip the *row group0*.

If there are multiple stripes in a file, we cannot guarantee that the *stripe*s in the cache table and in the raw data table have the same number of rows due to the difference in size of

Algorithm 3: Predicate Pushdown on Cache Table

Input : The Raw Data Table U, The Cache Table C, SearchArgument SA;

1 `InitializeReader` (U,C)

2 **Function** `InitializeReader`(u_{arg}, c_{arg})

3 **if** *cached column is filtered* **then**

4 $CacheReader \leftarrow$ initialize reader with SA on C;

5 $CacheRowGroups[] \leftarrow cr.getRowGroups$;

6 $PrimaryReader \leftarrow$ initialize reader on T;

7 $PrimaryReader.PrimaryRowGroups[] \leftarrow CacheRowGroups[]$;

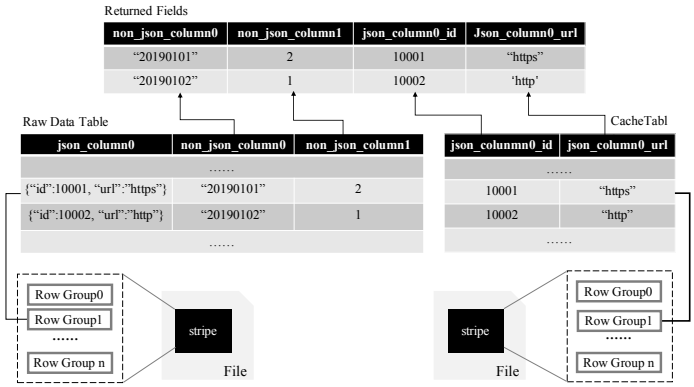


Fig. 10: One example of predicate pushdown

a row in the two tables. This would make us unable to perform *row alignment* when skipping the *row group*. However, as described in Section IV-C, each file in the cache table is parsed from the raw data table, and files with the same index have the same number of rows. Thus we only perform this optimization when a file has only one stripe and that is quite common because stripe size usually has default setting as 64MB or even larger.

V. Evaluation

A. Experimental Setup

We have implemented **Maxson** on Spark 2.3.0. We ran distributed Spark experiments on a 22-node cluster. Each node has two eight-core Xeon-2670 CPUs, 64GB memory, and one SAS disk, running Red Hat EnterPrise Linux Server release 6.2 (kernel 4.4.5-6) and JDK 1.8.0_171. To configure an environment that is close to a real production setup, we deployed HDFS, Yarn, and Hive services in high-availability mode. We also run Spark in yarn cluster mode. In the experiments, our tables are stored in Hive in *ORC* format. For each query, eight executors are allocated, each with 8GB memory size.

We obtained about 3 million analytic queries from Alibaba. To train our model, we randomly select 70% of them as training data set, 20% as validation data set, and the remaining 10% as test data set. We compare our predictor model with four baseline models:

TABLE II: 10 queries related descriptions used for experiments

SQL	JSONPath number	Property number in JSON	Nesting level	Average JSON size(Byte)
Q1	11	11	1	408
Q2	10	17	1	655
Q3	10	206	4	4830
Q4	1	215	4	4736
Q5	12	26	3	582
Q6	29	107	5	2031
Q7	3	12	2	252
Q8	5	17	1	368
Q9	1	319	3	21459
Q10	8	90	1	8692

- LR: It is a simple classification algorithm in machine learning.
- SVM: It is a popular baseline for classification algorithm in machine learning.
- MLPClassifer: It is a simple classification algorithm using neural network.
- LSTM: It is one of the most popular baseline for sequence labeling problems.

In the experiments of query performance, we do not attempt to replay all the queries in the Alibaba's production workload. The main reason is that it would require unreasonable amount of resources and time, not to mention running them multiple times to minimize variations and testing different scenarios to obtain an in-depth evaluation of various techniques. Therefore, we choose to run a sample set of representative queries to study the system's performance. Focusing on a sample of queries also allows us, as shown in our experiments, to perform in-depth analysis of various aspects of the proposed techniques within a reasonable amount of time. In Alibaba's production environment, queries are submitted by different users, and the queries from different users are typically independent on each other in the sense that they run on different subset of data due to data access control policies. Therefore, we select the queries issued by three representative users whose queries match the characteristics of most other users' queries. These queries take about 400 machine hours to complete, which is reasonable to perform our in-depth experimental analysis. Note that, while we only look at the performance of queries submitted by three sample users, we use the whole workload to train the prediction models. Therefore, the experiment results would indicate the actual improvement on query performance for a user in an actual environment. For a series of queries with spatial correlations, we only report the performance of one of them, so that it reflects the user's query performance over different tables.

The JSON description involved in each query can be seen in Table II. These queries involve a total of 10 tables. As the actual data values do not affect these queries performance, we synthetically generate 20 million rows of data for each table by following the real data hierarchies and formats. For each query, we repeat the execution 5 times and report the average execution time.

TABLE III: Comparison sequential features' importance of LSTM+CRF with different superivised machine learning algorithms and simple neural networks for prediction of JSONPath

Algorithm	Parameter	Precision	Recall	F1-Score
LR	multi_class='ovr', n_jobs=-1, penalty='l2', solver='newton-cg', max_iterations = 1000	1.0	0.397	0.568
SVM	multi_class='ovr', penalty='l2', loss='squared_hinge', max_iter=1000	1.0	0.559	0.717
MLPClassifier	solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(50, 10, 2), random_state=0	0.994	0.694	0.817
LSTM+CRF	numLayers=2, word_size=50, all_possible_transitions=True	0.985	0.912	0.947

TABLE IV: Comparison of LSTM+CRF and LSTM for prediction of JSONPath

Date Window Size	Model Name	Precision	Recall	F1-Score
1 week	LSTM+CRF	0.985	0.912	0.947
	LSTM	0.927	0.916	0.921
2 weeks	LSTM+CRF	0.997	0.975	0.916
	LSTM	0.912	0.889	0.9
1 month	LSTM+CRF	0.942	0.900	0.921
	LSTM	0.925	0.885	0.905

B. JSONPath Prediction

We use F1 score to evaluate the effectiveness of the predictor. In this experiment, we use the whole workload trace. First, we evaluate the importance of temporal features using the LR, SVM, MLPClassifier, and LSTM+CRF models. For each algorithm, the parameters are tuned to achieve the highest F1 score. As Table III shows, models that cannot take into account date sequences have particularly low recall and hence has a low F1 score. On the other hand, LSTM+CRF adds date sequence as a feature, which has a significant improvement on recall without lowering much the precision. Therefore, LSTM+CRF achieves a much higher F1 score. This experiment shows that temporal features are very important in our scenario.

Second, we examine the effectiveness of LSTM+CRF in handling temporal features. As a comparison, we use the LSTM module in Pytorch [27] to implement Uni-LSTM. We test different window size: one week, two weeks, and one month. For each window size, the parameters of LSTM+CRF and Uni-LSTM model are tuned to achieve the highest F1 score. The results of LSTM+CRF and Uni-LSTM are shown in Table IV. The F1 socre of LSTM+CRF is always higher than that of Uni-LSTM. With the window of 1 week, F1 scores are maximized for both models. It shows that LSTM is able to take into account the temporal features and capture the temporal patterns. The additional CRF layer, the probabilistic graph model, can learn the transition rules between labels, which are *MPJP* and non-*MPJP* in our case. As a result, LSTM+CRF has a higher F1 score in general for all window sizes.

C. Query Acceleration by Caching

In this experiment, we use four different cache constraints to examine our scoring function: 100GB, 200GB, 300GB, 400GB. We examine two strategies to choose *MPJP*s for caching under cache size constraints: 1) use the proposed scoring function to rank the *MPJP*s, 2) randomly select the *MPJP*s for caching. In addition, we also show the query performance without caching.

The total execution time is shown in Fig. 11, and the cached JSONPath for the ten queries can be seen in Table V (400GB is enough to accommodate all *MPJP*'s value). In general, it shows that a larger cache size can achieve shorter total execution time. This is because we can cache more *MPJPs* with a larger cache size to reduce more duplicated parsing overhead. Furthermore, the scoring strategy always outperforms random caching with different cache limits. When the cache size is enough to accommodate all *MPJP*s' values, random caching has the same performance as using our scoring function. From Table V, we can find that the scoring function tries to cache all *MPJPs* from the same query. For example, for the case with 300GB cache, all the *MPJPs* in Q2, Q3, Q4, Q6, Q9, Q10 are cached, while in contrast, with the random caching strategy, only part of the *MPJPs* in each query are cached. In addition, the scoring-based approach prefers to cache the *MPJP*s that have the higher *acceleration per byte*. For example, when the cache limit is 100GB, the *MPJPs* in Q10 are cached, which can accelerate the query by 45 times (see in Fig. 15).

To make a closer comparison, we break down the running time of two queries Q2, Q9 into three steps: Read, Parse, and Compute. The results are shown in Fig. 12. In Maxson, we eliminate the overhead of parsing by reading the cached value from the cached table. Besides, as shown in Fig. 12b and Fig. 12d, Maxson's input size is much smaller than Spark's input size, because Q2 and Q9's filtering predicates include properties within the JSON strings, so we can push such predicates down into the cache table to reduce the number of records that are read.

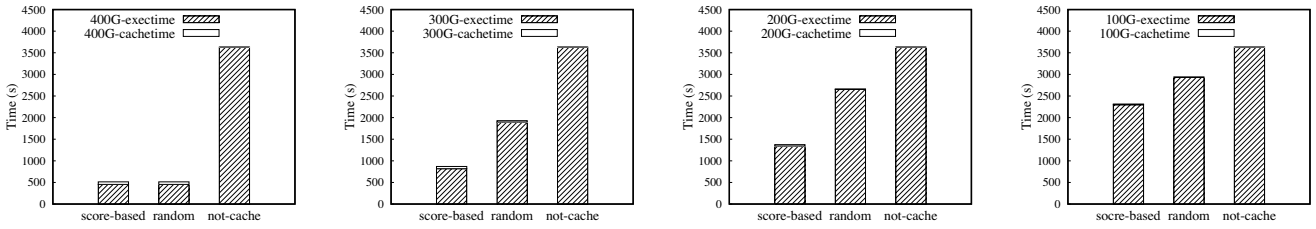
As for the overhead of caching, caching time with the scoring-based strategy is slightly longer than that with the random caching strategy. This is because the scoring-based approach attempts to cache *MPJPs* from more complex JSON strings, which can achieve higher query acceleration. On the other hand, the average cache overhead for each query only accounts for 1.7% of the execution time. This is because many queries with spatial relationship can share the cache overhead. In summary, the speedups achieved by Maxson are ranging from 1.5× to 6.5×.

D. Impact on Plan Generation

We examine the overhead of Maxson's process of modifying the physical query plan. Fig. 13 shows the time overhead for plan modification. We use the queries from Table II with the cache limit of 300GB, and record the time to generate the physical plans using Maxson and Spark, respectively. We find that Maxson is on average 0.4 seconds slower than SparkSQL.

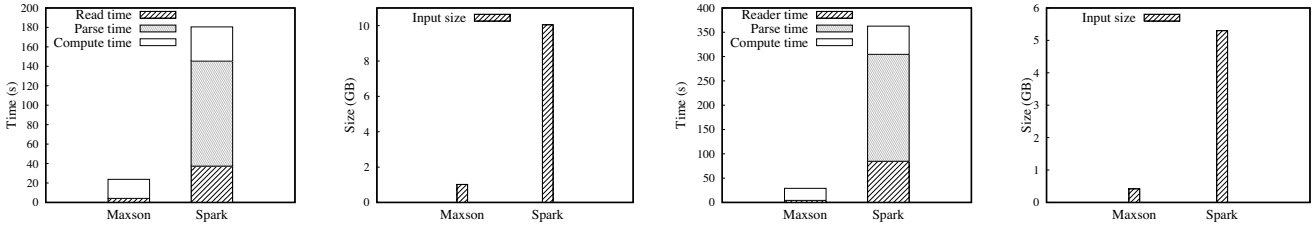
TABLE V: The cached JSONPath number in different cache limit: 400GB, 300GB, 200GB, 100GB

Query	400GB		300GB		200GB		100GB	
	Scoring Function	Random	Scoring Function	Random	Scoring Function	Random	Scoring Function	Random
Q1	11	11	0	7	0	4	0	3
Q2	10	10	10	6	0	4	0	2
Q3	10	10	10	8	10	6	0	3
Q4	1	1	1	1	1	0	0	0
Q5	12	12	0	8	0	5	0	2
Q6	29	29	29	23	17	14	7	8
Q7	3	3	1	1	0	1	0	0
Q8	5	5	0	3	0	2	0	1
Q9	1	1	1	1	1	1	0	1
Q10	8	8	8	6	8	4	8	2



(a) 400GB (b) 300GB (c) 200GB (d) 100GB

Fig. 11: Total execution for all ten queries with different cache constraints



(a) Q2 Breakdown (b) Input size of Q2 (c) Q9 Breakdown (d) Input size of Q9

Fig. 12: Breakdown of the query execution time, and input size

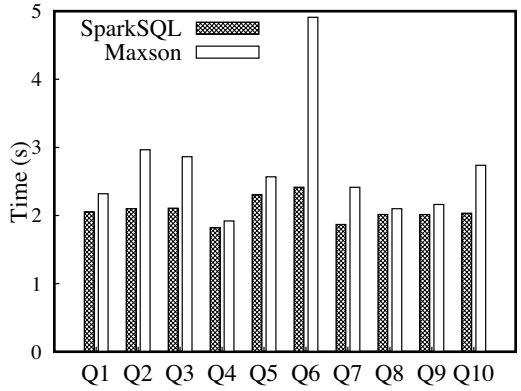


Fig. 13: Compare the time to generate SparkPlan for each query using SparkSQL and Maxson.

In general, the more JSONPath that are involved in a query, the longer it takes to generate the plan. However, the increase of planning time is negligible in comparing to the execution time of the entire job. The execution time of each query can be seen in Fig. 15.

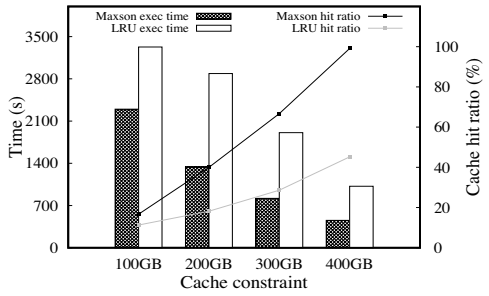


Fig. 14: Comparison of Maxson cache management and online cache management with LRU replacement strategy

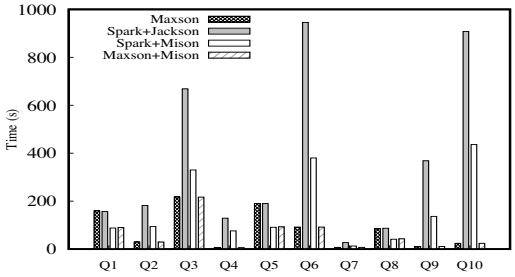


Fig. 15: Running time of ten queries from Table II using Maxson compared against Spark+Jackson, Spark+Mison, and Maxson+Mison

E. Comparison with Online Cache

In this experiment, we compare Maxson's prediction-based caching approach with an online caching with LRU policy. We run all queries mentioned in Section V-A in the original order in the workload trace. We record the total execution time and the cache hit ratio of the ten queries from Table II. As Fig. 14 shows, we can find that LRU policy has lower cache hit ratio and higher query execution time than Maxson. We analyze the cache access trace of the queries, and find that the LRU policy often evicts some cached values that can be used by other queries from different users. Furthermore, some queries with spatial correlation are often with similar submission time, hence there is little opportunity for the system to cache the values accessed by these queries. On the other hand, Maxson pre-parses and pre-caches these values before any of the queries is executed. Therefore, there is higher cache hit. Moreover, Maxson's scoring-based approach attempts to maximize the acceleration by selecting the JSONPaths with higher scores for caching.

F. Comparison with Other Parsers

This experiment is to examine the effect of using efficient JSON parsing methods, such as Mison [6]. This is to answer the question of whether it is still make sense to cache JSONPaths given that there are efficient JSON parsers.

We report the execution time of ten queries from Table II using Maxson, Spark+Jackson, Spark+Mison, and Maxson+Mison respectively. Jackson [28] is the default JSON parser in SparkSQL. We use Pirkkr [29], the most popular open source implementation based on the paper of Mison. We set Maxson's cache limit as 300GB. As shown in Fig. 15, due to its high efficiency of JSON parsing, Mison can reduce query execution time significantly, especially in Q6 where the JSON pattern has little change in the dataset.

However, as Mison still has the overhead of projecting data fields, which is especially significant when the JSON schema varies significantly within the dataset. Therefore, we can see that for Q2, Q3, Q4, Q6, Q7, Q9, and Q10, the caching in Maxson is more effective in eliminating the parsing overhead, which is unaffected by the variation of JSON schema in the dataset. For queries like Q1, Q5, and Q8, Maxson does not cache their JSONPaths, therefore, Mison can be a good complementing technology to speed up the parsing of uncached JSONPaths.

VI. Related Work

Analysis System of Raw Data. Driven by practical needs of data analytics tasks, a lot of recent researches have been focusing on building system to support querying directly on raw data to eliminate the cost and evade the difficulties of modeling, schematizing, transforming, and loading the data. [30], [31] support directly querying JSON document within RDBMS by modifying the RDBMS storage layer. In this way, RDBMS users can store, query, and index JSON data without the need to incorporate JSON data into tables with rigid schemas, which is often hard to achieve. [32], [33] support multiple storage layouts for different access pattern over raw data, and attempt to adapt the storage according the changes of workload. NoDB [34] builds indices of attributes incrementally over data in CSV format to navigate as close as possible to the position of the queried data. Karpathiotakis et al. [35] propose JIT-compiled access path to adapt the query engine according to the formats of raw data. Slalom [36] made on-the-fly partitioning and indexing to access raw data efficiently. While this category of work focuses on developing querying and data access techniques to speed up raw data access, the pre-caching approach adopted by Maxson focuses on eliminating duplicate overhead parsing raw data by caching the parsed values in advance. This pre-caching technique can be incorporated into raw data processing engines in a complementary manner.

Optimized Parser of Semi-Structured Data. Mison [6] uses SIMD instructions to build a *field index* for a JSON string based on special characters, such as brackets and colons. It can perform efficient projection without de-serializing JSON strings completely. Sparser [7] adopted a new approach that applies filters on the raw byte stream of data before parsing. It is based on the observation that many real-world applications are highly selectivity. Other examples of optimized JSON parsers include Gson [8] and RapidJSON [9]. While these approaches can effectively reduce the overhead of JSON parsing, Maxson is designed to address an orthogonal problem: duplicate JSON parsing by queries with correlations.

Parsing performance is also an issue for other semi-structured data formats like XML [37]. Many techniques (eg. [37]–[40]) have been proposed to accelerate XML parsing. Maxson's pre-caching technique can also be applied to other data formats, such as XML.

Predictive Caching. Apollo [11] and Promise [14] exploit query patterns in a session to predictively cache query results. While Apollo [11] is based on query's transitivity to predict next query and Promise [14] focuses on navigational queries. Maxson does not explore the pattern of a series of queries in a session to predict the next query, but relies on the temporal and spatial correlations of queries to cache repetitive JSONPaths. Lempel et al. [12] cache search results based on a probabilistic model of users of search engines. It prioritizes the cached pages based on the number of users who are currently browsing the pages. Ozcan et al. [13] propose caching strategies for search engines, which are cost-aware and consider both static and dynamic setups. Maxson also adopts a scoring function to prioritize caching JSONPaths according to the amount of query acceleration.

VII. Conclusion

This paper focuses on optimizing query performance in data analytics systems that directly store and query JSON data. By analyzing a real trace of production workload, we identify temporal and spatial correlations among different queries in terms of JSONPath access. Such correlations would incur redundant parsing of JSON data to achieve the same data values. We propose Maxson, a JSONPath caching system that is implemented as a component in SparkSQL. Maxson

conducts daily predictions of JSONPath access by user queries, and performs pre-parsing and pre-caching when the system resource is under-utilized, typically during mid-night. As shown by experiments on a real large-scale workload trace, Maxson's hybrid LSTM and CRF prediction model performs very well in both precision and recall in comparing to other alternative models, and the proposed scoring function is able to prioritize caching JSONPaths to optimize the effectiveness of caching. Furthermore, Maxson is able to modify SparkSQL query plans to efficiently query both cached and uncached data, as well as to support predicate pushdown to optimize query performance.

VIII. ACKNOWLEDGGMENTS

We thank Lu Lu and Yufen Zong for their valuable comments. This work is supported by the National Key Research and Development Plan (No. 2017YFC0803700), NSFC (No. 61772218, No. 61832006 and No. 61802140), Hubei Provincial Natural Science Foundation (No. 2018CFB200) and Alibaba Innovative Research (AIR) program.

REFERENCES

[1] "XML VS JSON." http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html#software.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[4] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 147–156.

[5] "Apache Drill." https://drill.apache.org/.

[6] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, "Mison: a fast json parser for data analytics," *in Proceedings of the VLDB Endowment*, vol. 10, no. 10, pp. 1118–1129, 2017.

[7] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia, "Filter before you parse: Faster analytics on raw data with sparser," *in Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1576–1589, 2018.

[8] "Google Gson." https://github.com/google/gson.

[9] "Rapidjson." https://rapidjson.org.

[10] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: Batched stream processing for data intensive distributed computing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2010, p. 63–74.

[11] B. Glasbergen, M. Abebe, K. Daudjee, S. Foggo, and A. Pacaci, "Apollo: Learning query correlations for predictive caching in geo-distributed systems," in *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, 2018, pp. 253–264.

[12] R. Lempel and S. Moran, "Predictive caching and prefetching of query results in search engines," in *Proceedings of the 12th International Conference on World Wide Web*. ACM, 2003, pp. 19–28.

[13] R. Ozcan, I. S. Altingovde, and Ö. Ulusoy, "Cost-aware strategies for query result caching in web search engines," *ACM Transactions on the Web (TWEB)*, vol. 5, no. 2, p. 9, 2011.

[14] C. Sapia, "Promise: Predicting query behavior to enable predictive caching strategies for olap systems," in *Proceedings of International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2000, pp. 224–233.

[15] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive-a petabyte scale data warehouse using hadoop," in *Proceedings of 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 2010, pp. 996–1005.

[16] "Maxcompute." https://www.alibabacloud.com/zh/product/maxcompute.

[17] "Nobench benchmark." https://quickstep.cs.wisc.edu/argo/nobench.tar.bz2.

[18] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE Computer Society, 2010, pp. 1–10.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the Eighteenth International Conference on Machine Learning*, 2001, pp. 282–289.

[21] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, no. 1, pp. 19–67, 2005.

[22] G. D. Forney, "The viterbi algorithm," *in Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[23] S. H. Walker and D. B. Duncan, "Estimation of the probability of an event as a function of several independent variables," *Biometrika*, vol. 54, no. 1-2, pp. 167–179, 1967.

[24] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[25] "Multi-layer Perceptron." https://scikit-learn.org/stable/modules/neural_networks_supervised.html/.

[26] "Apache ORC." https://orc.apache.org.

[27] "Pytorch." https://pytorch.org/.

[28] "Jackson." https://github.com/FasterXML/jackson.

[29] "Pikkr." https://github.com/pikkr/pikkr.

[30] C. Chasseur, Y. Li, and J. M. Patel, "Enabling JSON document stores in relational systems," in *Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013, New York, NY, USA, June 23, 2013.*, 2013, pp. 1–6.

[31] Z. H. Liu, B. Hammerschmidt, and D. McMahon, "Json data management: supporting schema-less development in rdbms," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 1247–1258.

[32] I. Alagiannis, S. Idreos, and A. Ailamaki, "H2o: a hands-free adaptive store," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 1103–1114.

[33] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki, "Here are my data files. here are my queries. where are my results?" in *Proceedings of 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[34] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," in *in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 241–252.

[35] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki, "Adaptive query processing on raw data," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1119–1130, 2014.

[36] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki, "Slalom: Coasting through raw data via adaptive partitioning and indexing," *Proceedings of the VLDB Endowment*, vol. 10, no. 10, pp. 1106–1117, 2017.

[37] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. P. Popowich, "Parallel scanning with bitstream addition: An xml case study," in *Proceedings of the 17th International Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2011, p. 2–13.

[38] C. Koch, S. Scherzinger, and M. Schmidt, "Xml prefiltering as a string matching problem," in *Proceedings of 2008 IEEE 24th International Conference on Data Engineering*. IEEE, 2008, pp. 626–635.

[39] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing xml streams with deterministic automata and stream indexes," *ACM Transactions on Database Systems (TODS)*, vol. 29, no. 4, pp. 752–788, 2004.

[40] T. Takase, H. Miyashita, T. Suzumura, and M. Tatsubori, "An adaptive, fast, and safe XML parser based on byte sequences memorization," in *Proceedings of the 14th International Conference on World Wide Web.*, 2005, pp. 692–701.