DE GRUYTER OPEN

Int. J. Appl. Math. Comput. Sci., 2017, Vol. 27, No. 4, 713–726
DOI: 10.1515/amcs-2017-0049

amcs

EFFICIENT STORAGE, RETRIEVAL AND ANALYSIS OF POKER HANDS: AN ADAPTIVE DATA FRAMEWORK

MARCIN GORAWSKI [a,*], MICHAL LOREK [a]

[a]Institute of Informatics
Silesian University of Technology, Akademicka 16, 44-100 Gliwice, Poland
e-mail: {marcin.gorawski,michal.lorek}@polsl.pl

In online gambling, poker hands are one of the most popular and fundamental units of the game state and can be considered objects comprising all the events that pertain to the single hand played. In a situation where tens of millions of poker hands are produced daily and need to be stored and analysed quickly, the use of relational databases no longer provides high scalability and performance stability. The purpose of this paper is to present an efficient way of storing and retrieving poker hands in a big data environment. We propose a new, read-optimised storage model that offers significant data access improvements over traditional database systems as well as the existing Hadoop file formats such as ORC, RCFile or SequenceFile. Through index-oriented partition elimination, our file format allows reducing the number of file splits that needs to be accessed, and improves query response time up to three orders of magnitude in comparison with other approaches. In addition, our file format supports a range of new indexing structures to facilitate fast row retrieval at a split level. Both index types operate independently of the Hive execution context and allow other big data computational frameworks such as MapReduce or Spark to benefit from the optimized data access path to the hand information. Moreover, we present a detailed analysis of our storage model and its supporting index structures, and how they are organised in the overall data framework. We also describe in detail how predicate based expression trees are used to build effective file-level execution plans. Our experimental tests conducted on a production cluster, holding nearly 40 billion hands which span over 4000 partitions, show that multi-way partition pruning outperforms other existing file formats, resulting in faster query execution times and better cluster utilisation.

Keywords: big data, storage model design, data architecture, data access path optimization.

1. Introduction

Over the past fifteen years, online poker has undergone a remarkable transformation and shown an unimaginable growth in popularity. In the late 1990s only two online poker sites operated, attracting mainly recreational players. What was once a game for few hobbyists became one of the key segments of the multi-billion dollar gambling industry.

As the business grows organically, the poker ecosystem becomes more complex and difficult to influence or understand. The traditional performance indicators such as daily registrations, total deposits, the number of unique players and the number of hands played no longer reflect the true nature of the ecosystem's dynamics and social interactions involved. Inevitably, this leads to a significant loss of control over the site and in extreme cases it can drive some groups of players away, causing serious destabilisation of the poker economy.

To gain and maintain a competitive advantage in the market, the business is forced to invest in the research and development of new strategic management tools allowing better understanding of all the social and economic mechanisms present in the poker environment. Thus, through detailed and comprehensive analysis and modelling of poker hands, the business can recognise many behavioural patterns and translate them into player focused strategies that shape a healthy and well-balanced poker ecosystem.

Since hand data describe all of the player actions in the poker game, analysing the hands can reveal details about player behaviour that are interesting to the business. There are many standard metrics which can be used to profile and classify a poker players playing style

*Corresponding author

(AF, PFR, VP$IP, W$SD[1], etc.). An ability to quickly and efficiently analyse hand data can help the business to guide and evaluate the performance of marketing initiatives and promotions.

Online poker businesses will often run promotions to encourage players to play on their site. Many of these types of promotions have the potential to alter the behaviour of players in ways that might be detrimental to the poker ecosystem. For example, they might encourage recreational players to play weaker hands, and the implications of this poorer play could lead players to lose their money, have a bad experience and ultimately discontinue their play. Deep hand analyses can reveal if incentives are targeting the intended segment of the player base and help understand the impact of the promotion on the game itself.

Unlike other online games of chance such as roulette or slots, mainly due to its specific game structure, poker is more susceptible to fraudulent activities such as collusion and chip dumping. Depending on the form of the committed fraud, business needs to employ various counter-strategies to maintain game integrity and security. For example, a typical approach to detect instances of bonus abuse is to identify a pattern of money movements between newly registered players and existing accounts. In many cases, it can be observed that an individual or a group of people open multiple accounts to collect a bonus which is subsequently lost (through a poker game play) to players participating in the scheme. Another form of poker fraud, collusion, occurs when a group of players cooperate together using predefined signals in order to take advantage of unsuspecting players competing against them.

However, many forms of collusion can also be mitigated using information stored in hands by exposing decisions that would only be considered correct if a player had unfair access to additional poker information. In a game of imperfect information such as poker, a player attempts to make the most profitable decision based on the information that has been naturally revealed to them over the course of a hand. Collusion grants additional information unfairly to the cheating player, and alters their decisions. If a player is found to be making a profit while consistently making seemingly incorrect decisions, we may conclude that the player has unfairly received additional information.

Although extremely difficult, detailed analysis of hands helps uncover the use of decision support software that provides an unfair advantage to the player by providing information that they would not normally observe through their own game play. This is manifested through variations in the play style of a player, however, the player would need to play a large amount of hands for us to have a high degree of confidence that this is a variation in a play as opposed to simple variance.

In addition, one of the common problems online poker rooms face is the use of computer bots. Bots are mainly used to exploit certain types of promotions, where players are rewarded for their cumulative gaming activity volume. Some bots have been created that are outright winners in today's poker games. Usually, the detection of bots is achieved by careful analysis of a player's actions stored in hands. Consistent playing patterns in identical situations over a large number of hands, or across multiple accounts, can prompt an investigation since humans tend to get fatigued and make minor mistakes and variations. Bots also have non-human attributes that can be identified, such as timings and mouse movements.

Being subjected to strict international anti-money laundering regulations and heavily controlled within local gambling markets, poker companies realised the great importance of the information contained within the hands and started archiving them to prevent the crime and provide fair gaming environment. Storing the ever increasing number of hands as well as development of new poker variants and continuous surge of poker popularity have led to enormous data explosion. So far, poker companies have relied heavily on relational databases to build and maintain data-banks consisting of historical hands. However, with tens of millions of hands played daily, storing them efficiently has become a big technical challenge to traditional databases systems. Equally, fast hand data retrieval and its complex analysis pose additional difficulties such as slow data extraction, lack of parallel computational capabilities and inherent language constraints. To overcome these limitations, many companies start considering other emerging technologies such as Hadoop to deliver better performance and scalability.

2. Related work

In recent years online poker has drawn the interest of machine learning and data mining researchers. Their work has demonstrated that by utilising machine learning it is possible to model an opponent by analysing their actions (Mealing and Shapiro, 2015). During the game play, such a model improves player decisions by presenting a range of suitable strategies against other players (Teófilo and Reis, 2011). Other research work has shown that by mining the poker hand data sets and using statistical probabilities new advanced poker tactics can be defined (Ambekar *et al.*, 2015). Another area of poker research concentrates on defining and developing profitable online poker playing agents (bots) (cf. Teófilo *et al.*, 2013; 2014). In general, it can be argued that the main purpose of online poker research is to identify an optimal poker strategy

[1]AF: aggression factor, PFR: pre flop raise, VPIP: voluntarily put money into pot, W$SD: won money at showdown.

and ultimately, through an assisted or automated play, maximise the player's profits (Miltersen and Sørensen, 2007). For this reason, there is a lot pressure on poker business to assure that the use of bots or decision supporting systems is correctly detected and eliminated in online card rooms.

In terms of research dedicated to optimising the performance of data retrieval in Hadoop, our work can be considered an alternative to the existing hybrid solutions. The idea behind hybrid data solutions is to combine the best features of Hadoop and the RDBMS together (Alamoudi *et al*., 2015; Abouzeid *et al*., 2009). In our work, we try to bring many of the existing DB features and incorporate them into Hadoop without utilising database systems as auxiliary tools to support them. There are many optimisation techniques that allow increasing the performance of MapReduce based solutions (Jiang *et al*., 2010; Thusoo *et al*., 2010). For instance, one way to improve Hadoop's performance is to design an optimised storage model that utilises various indexing solutions (Richter *et al*., 2014). Although the use of column oriented storage formats (RCFile, 2016; ORC, 2016) dominates the design of modern Hadoop-based data warehousing solutions, we present a storage model that is both read optimised and row based.

3. Background

Knowing all the traditional DB limitations as well as trying to minimise licensing fees and simplify maintenance procedures, the business has decided that a new research project needs to identify a suitable design for a system that will replace the existing DB platform with Hadoop. Thus, we have been tasked to investigate how to migrate the entire hand history into the Hadoop based platform while ensuring the functionality of all the existing legacy subsystems.

3.1. Poker hand object anatomy. In online gaming, poker hand objects are one of the most important and fundamental units of the game state, and can be considered logical containers comprising all the events that pertain to a single hand played. Hand-related events are generated by one of the many game servers during a game play, and usually, once the hand is finished, they are collated by an intermediate buffering service to form a complete poker hand object, before it is sent further for the actual storage. Depending on the complexity of a particular game system, the number of events kept inside the hand object can vary, but typically their structure defines up to a hundred of different event types. Each individual hand object is independent of others and is sufficient to reconstruct the actual and complete game play. Before hand objects are persisted to the actual archiving platform, they need to be first decomposed according to one of the chosen storage strategies.

One of the common procedures to decompose a poker hand object is to model the information stored inside the object and define a set of normalised entities to highlight all the relationships between them. Figure 1 depicts a simplified hand object's entity relationship diagram. As a result, a number of physical tables containing the hand's data are created. However, storing event data in separate tables introduces significant storage overhead associated with row and page headers. Depending on the database storage architecture, the overhead size can vary and ranges from 6 to 27 bytes per stored row (PostgreSQL, 2016; Delaney, 2009; Mullins, 2000).

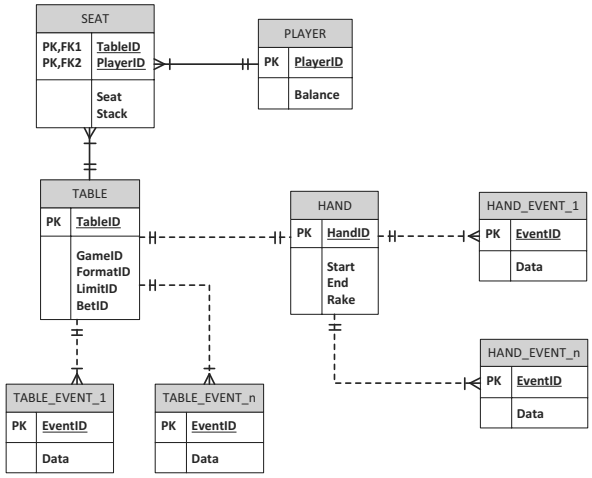

Fig. 1. High level conceptual data model of a poker hand.

Furthermore, a normalised data model significantly increases retrieval costs when hand objects need to be reconstructed. To obtain the full representation of the hand object, multiple join operations are required. In addition, the development of new poker game features can lead to an incoherent hand model and cause confusion when querying logically diffused tables.

The total size of the overhead incurred by the multi-entity design can be estimated and expressed using the following equations:

$$r_o = r_{hdr} + e_{sz}, \tag{1}$$

$$O_1 = N_h r_o(\overline{N}_{pe} + \overline{N}_{te} + \overline{N}_{he}), \tag{2}$$

where N_h denotes the total number of poker hands and r_o represents the overhead associated with both the row header and row offset array. Meanwhile, $\overline{N}_{pe}$, $\overline{N}_{te}$ and $\overline{N}_{he}$ represent respectively the average numbers of user, tournament and hand events stored per hand. The symbol e_{sz} denotes the size of a single entry in the row offset array.

3.2. Existing solution. The entire existing archived data hosts nearly 40 billion hands containing over 2.4 trillion game events. On the average, the length of the serialised and compressed hand object (*blob*, $\overline{b}$) is 715 bytes. Such a structure includes a poker hand's every hand and table event, and offers a better way of collecting the information scattered between many entities. The opaque nature of the hand blob binary format does not impose any special requirements for their storage. Typically, any database data type that allows storing a collection of bytes is sufficient (e.g., *VARBINARY*, *BYTEA*, etc.). To simplify data management, blobs are physically stored according to a partitioning scheme based on the value of the *day_key* column, derived from the *hand_endtime* hand attribute. However, seasonal hand volume fluctuations result in considerable differences in partition sizes.

In contrast to the heavily normalised approach, the use of blobs drastically reduces storage overheads, but it does not eliminate them completely. In fact, when logical ordering is required, placing the records of a variable length on a fixed size page results in very poor page utilisation. Since the measured standard deviation of the blob's length is 170, using a 99% confidence interval, we can see that a single sample could vary $\pm$ 438.65 from the mean.

Thus, the average blob length $\overline{b}$ cannot be used to closely estimate the amount of the overhead associated with the unused space on a page. For example, if we consider a page of size P_{sz} and include both page and row related overheads (P_{hdr}, r_o), then only P_r rows can fit on that page:

$$P_r = \left\lfloor \frac{P_{sz} - P_{hdr}}{r_o \overline{r}} \right\rfloor, \tag{3}$$

where $\overline{r}$ denotes the average row length. Hence, the average amount of unused space on a single page o_p is expressed by

$$o_p = (P_{sz} - P_{hdr}) - P_r r_o \overline{r}. \tag{4}$$

Depending on the chosen DB platform and the definition of the blob storing row, the values of P_r and o_p vary. Given the existing PostgreSQL based cluster, we can examine two scenarios. One assumes that the definition of the table storing blobs does not include any additional columns. In this case, 11 blobs of average length can be placed on a page resulting in 99.63% page utilisation (30 bytes of unused page space). The more practical case includes a column required for partitioning purposes and a column needed to guarantee a logical order of the rows—altogether an extra 10 bytes per row stored. Such an adjustment decreases the number of rows per page to 10 and reduces the average page utilisation to 91.79% (672 bytes of unused page space). Nonetheless, the real scale of the problem still remains obscured. Figure 4 shows blob length distribution for the entire set and compares it with distributions obtained for five equally distanced partitions.

In order to accurately investigate the actual value of the overhead, we took a random sample of 400 days out of the total population. For each selected day, we measured the number of pages used together with the mean, median and standard deviation of the overhead incurred. Figure 2 shows how these values distribute over time for both cases mentioned above. We also calculated the standard deviation of a weighted mean based on the number of pages. We determined that the sampled daily standard deviation of the overhead remains relatively high at about 250. This indicates an extremely wide range of the overhead values ($\pm$ 645). Meanwhile, the weighted mean is 400 and its standard deviation is 20, implying a certain degree of the average overhead stability over time, and offers a better picture of the issue.

Consequently, the total amount of the overhead associated with the unused space on page (P_f) can be expressed using the following formulae:

$$O_2 = N_h r_o (2 + \overline{N}_{pl}) + \sum_{i=1}^{N_p} N_{f_i}, \tag{5}$$

$$P_f = P_{sz} - \sum_{j=1}^{N_r} |r_j| - P_{hdr} - P_r r_o, \tag{6}$$

where $\overline{N_{pl}}$ denotes the average number of players per hand, N_f represents the unused space on a page and $|r_j|$ is the length of the j-th row on a page. Our calculations revealed that the unused page space makes up more than 5% of the entire size of all blobs.

On the plus side, the adoption of a new simplified logical data model decreases the effort required to retrieve and reconstruct a hand object. Figure 3 shows an enhanced version of the original entity diagram. Apart from the main table dedicated to the blobs, only a small number of attributes are kept in two ancillary tables. This allows the maintenance of the partition scheme and creation of the necessary indices.

Considering all the changes mentioned above, the performance of the retrieval of blobs is greatly improved. However, the expense of increased processing efforts is related to blob decompression and deserialisation. Thus, multiple and complex JOIN operations can be replaced with a trivial SQL statement specifying the necessary search conditions. Furthermore, the development and use of iterators (blob decoding routines) allow selective access to the information inside the blob.

Listing 1 shows one possible way of accessing all hands of a particular type played between two specific players using a new data model (Fig 3).

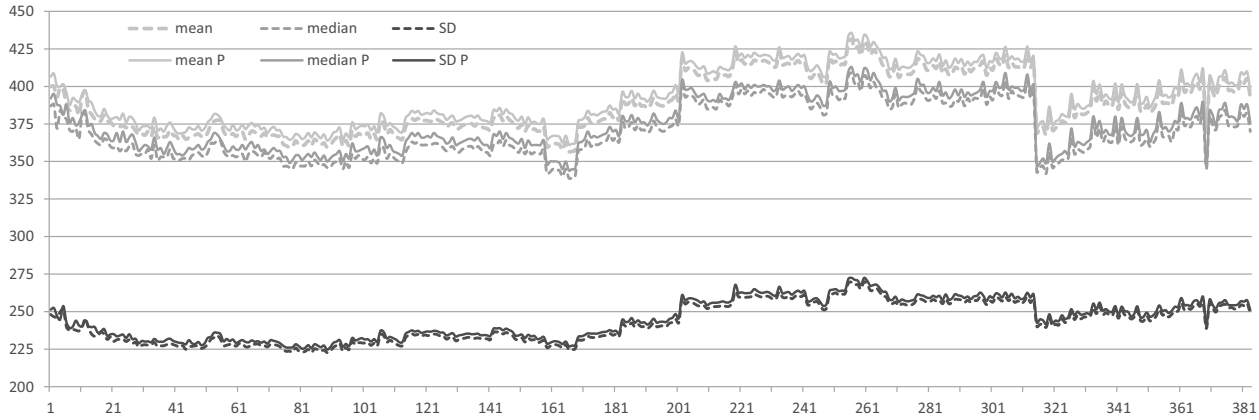

Fig. 2. Distribution of the page overhead for partitioned and non-partitioned tables.

Listing 1. Finding all the hands played between two players over particular game type.

```
SELECT DECODE_HAND(d.hand_data, feature_list)
FROM hand_data as d
WHERE d.hand_key IN (
    SELECT hand_key
    FROM hand_player as p1
    WHERE player_key = @player_key_1
    INTERSECT
    SELECT hand_key
    FROM hand_player as p2
    WHERE player_key = @player_key_2
)
INNER JOIN hand_info as i
    ON d.hand_key = i.hand_key
WHERE i.game_type_id = @game_type_id;
```

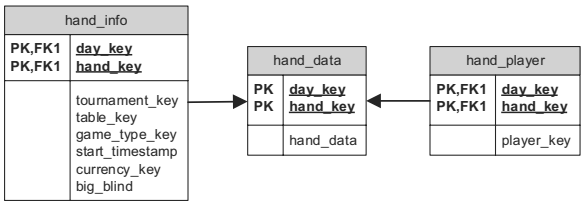

Fig. 3. Compact version of the original entity diagram.

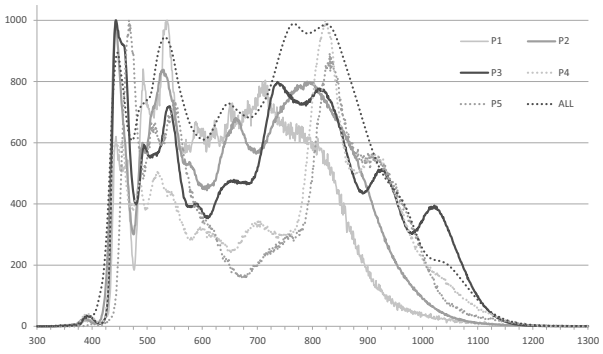

Fig. 4. Comparison of blob length distributions.

3.3. Hadoop storage formats. Unlike traditional databases storage engines, with Hadoop, data are maintained by splitting the files into blocks and storing them across multiple HDFS data nodes (Hadoop, 2014). The number of nodes storing the same block depends on the chosen replication factor and by default is set to 3 (HDFS, 2016). The location of all the blocks is maintained by a dedicated node called NameNode. When a client application tries to open an existing file, NameNode provides the list of all its data block locations together with the information which data nodes host them. In a situation where a new file is created on the HDFS, NameNode coordinates the writing process by recording the final position of all the successfully distributed blocks between data nodes. The read and write data access is achieved through specialised HDFS libraries. There are two options for obtaining data in a MapReduce application. These include implementing the InputFormat interface, or extending the supplied input formats such as FileInputFormat.

A Hadoop user has a choice on how to format and persist data in the HDFS. This is in contrast to the relational databases, where one is forced to use only one storage model that is constrained by the internal database engine architecture. Although some DB vendors facilitate the development of customised storage engines (MySQL, 2016), the complexity of database query engines restricts the independence of the storage model definition of the engine itself. The fact that Hadoop does not enforce explicitly any particular storage format on the user, along with the flexibility of the storage format API, makes Hadoop a very convenient and extremely flexible distributed data storage platform. Consequently, several additional storage formats such as RCFile and ORC have been proposed for Hadoop, featuring a read-optimised columnar data layout as well as advanced data encoding and compression features.

As Hadoop has gained greater recognition over time, many new solutions based on MapReduce and the HDFS have been introduced to the Hadoop ecosystem. In many cases, the main purpose of these new products was to address MapReduce interface complexity and accommodate many of the existing RDBMS features into Hadoop. This was achieved by the ability to access and modify data using query languages, logical data organisation into relational structures, etc.

Hive is a good example of such an application that allows SQL-trained users to query data stored in the HDFS using HiveQL (Thusoo *et al*., 2010; Hive, 2014). Hive queries work in a way that is similar to traditional databases. They are parsed, type-checked and optimised before the physical plan is generated. Subsequently, they are executed in the form of multiple map/reduce tasks using Hadoop. Given the inherent horizontal scalability, fault tolerance and better control over storage related overheads, Hive can be considered a viable alternative to relational databases.

In terms of hand archive migration and storage, the real benefit of Hive is the flexibility in table structure declaration and lack of restrictions in the number of ways data can be formatted. Hive supports both primitive (integers, string, dates, etc.) and complex (arrays, maps, structs) data types. The ability to declare an ARRAY column in a table definition enables us to maintain a list of players in a single column along with the hand blob. In addition, all the attributes from the *hand_info* table can also be included as part of the hand table definition. A revised version of the optimised data model for Hive is presented in Fig. 5. By reducing the number of tables to one, we completely eliminate the need to perform JOIN operations and further simplify queries accessing the blobs. Even though JOIN operations are supported by Hive, serious performance degradation can be observed in situations where large data sets of comparable cardinality are combined together. Finally, Hive's intrinsic reliance on MapReduce as an execution engine guarantees computational scalability and enables great freedom of choice in designing an efficient storage format that satisfies our data access and processing needs.

hand_data	
PK	day_key
PK	hand_key
	tournament_key
	table_key
	game_type_key
	currency_key
	small_blind
	big_blind
	player_keys[ ]
	hand_data

Fig. 5. Revised version of the hand table definition for Hive.

3.4. Challenges. Logical and physical design considerations of the next generation of poker hand data storage and processing platform have exposed some fundamental weaknesses of both the relational databases and Hadoop-based solutions such as Hive or MapReduce. In the context of storage-related inefficiencies, we perceive page-oriented data allocation as a severely restrictive factor that introduces large and difficult to reduce space overheads. As pointed out in Section 3, page and row maintenance costs, along with logical order enforcement and partitioning schemes, can negatively affect storage utilisation and in reality rule out the use of the RDBMS as a feasible solutions. Meanwhile, the biggest challenge we face with Hadoop is that it was designed to be a batch-oriented data processing platform. Although it completely separates the underlying storage model from the programming paradigm and allows handling various types of data, in principle it is not optimised to support highly selective queries. For example, when a MapReduce job or HiveQL query is run against a large data set consisting of thousands of large files, and we need to position and retrieve only few rows matching the search criteria, the entire file set needs to be accessed and processed. Naturally, this presents a major obstacle, as every single data search request could consume the resources of the whole cluster. This can potentially block others users from submitting their queries and run them simultaneously.

One way to address this issue is to introduce physical data segregation by organising logically correlated files into folders using arbitrary, application-driven rules (i.e., creation date). By reducing the number of files processed in a single query, more jobs could share the cluster's computational resources and run in parallel, resulting in lower user query latency and higher cluster occupancy. Unfortunately, in many Hadoop-based corporate environments that maintain and analyse hundreds of terabytes of raw data, complex folder structure reorganisation can be insufficient. If we imagine a situation where a company generates 100 gigabytes of data on a daily basis and exposes them to the end users through the MapReduce or Hive interface for further analysis, we can see that physical file partitioning does offer a reasonably scalable solution. Thus, in order to bring some resource control into the Hadoop environment and enable an improved multi-tenancy application support, the concept of scheduling policies has been introduced into YARN (YARN, 2016). However, this approach does not improve query latency or minimise the

volume of data to be processed. In essence, it ensures that the available cluster resources will be balanced between concurrent applications according to the chosen scheduling policy.

In order to improve query responsiveness, some of the existing data indexing algorithms have been explored and integrated into Hadoop. Primarily, they have been exploited by MapReduce jobs, and in the case of HAIL required decorating map tasks with customised annotations, specifying the selection predicates and the list of projected fields (Richter *et al*., 2014). Unfortunately, no indexing enhancements to the MapReduce interface are visible to the applications that conform to the original interface such as Hive or Spark. In practice, they require extensive development effort in order to benefit from them. Equally, the indexing features that Hive offers are functionally incompatible with the MapReduce processing model, limiting their use by the other Hadoop components. In our case, it is crucial that the hand querying capabilities offered by the data archiving platform allow transparent data indexing regardless of the chosen Hadoop applications.

Finally, it can be argued that the main downside of the HDFS file management is its lack of random write access to the disk (Shvachko *et al*., 2010). Once a file is copied to HDFS, it can only be read from or appended to. Writing to an existing file at an arbitrary position is not supported. In order to modify any part of a file stored in the HDFS, the file has to be first removed and uploaded again in a modified form. Therefore, these restrictions will have a detrimental effect on the design of an incremental hand loading procedure and will affect processes like file content sorting and index updates.

4. Adaptive data framework

Having understood the poker hand's unique life-cycle and analysed the internals of MapReduce and the HDFS, we decided to design a dedicated data storage platform based on Hadoop. We focused on maximising the performance of poker hand retrieval, reducing the operational costs related to hand storage and ensuring the portability of the proposed storage model in the Hadoop environment.

4.1. Overview. The proposed adaptive data framework (ADF) comprises a number of dedicated components performing actions related to the data access path shaping including data loading and filtering. Figure 6 presents a high level view of the ADF architecture and highlights the main relations between its components. The key parts of the ADF are *Partition Pruner*, *Index Reader*, *Segment Splitter* and *Predicate Handler*. Each of the components employs various optimisation techniques during the physical data access phase. The strength of the ADF lies in its flexibility to accommodate new data access optimisation features at a component level without sacrificing the integrity of the entire framework. In the following sections, we will look more closely at each of the components and describe the role they play in the ADF.

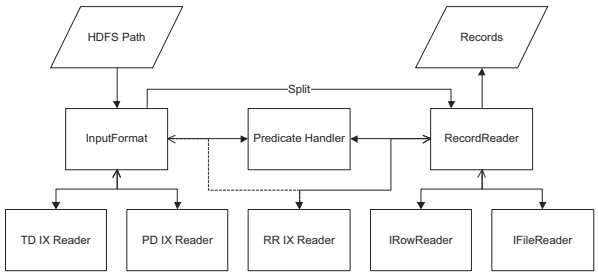

Fig. 6. Simplified model of the ADF.

4.2. Contributions. The presented adaptive data framework is our response to the obstacles mentioned in Section 3.4. The following describes the main contributions of our work:

Dynamic partition elimination. In order to control the poker hand search space, we introduce a list of dynamic partition elimination strategies based on expression tree decomposition and supported by a range of global indexing structures.

Complex structure indexing. We provide the means to index the information stored in complex data type structures so that the other ADF components can benefit from the optimised data access path and reduce poker hand retrieval costs.

Cost based workload balancing. At the storage level, we introduce a workload balancing algorithm based on predicate-triggered index evaluation. This is responsible for preparing the most efficient hand rows accessing strategy.

Uniform storage model. We offer a storage model that conforms to the standard MapReduce interface and enables end users to access poker hand archives through the traditional MapReduce jobs, HiveQL statements against the tables defined in Hive, or via Spark Data-Frames.

Performance evaluation. We evaluate the proposed data framework by measuring its performance in a controlled production environment. In addition, we present performance comparison results of our solution and other popular approaches.

4.3. Index-based partition elimination. In general, a common approach to improve a data-driven system's performance is to reduce the volume of data, which

otherwise have to be examined needlessly by a query. This reduction is achieved through physical data reorganisation, usually according to some partitioning dimension, such as date. However, such an approach does not provide enough flexibility to support a wide range of queries that are date agnostic. For example, while querying the hand table for all the hands that were completed within the last 10 days, we can benefit from the date-based partition scheme. Conversely, if we want to find all the hands of a particular player, such a scheme does not offer any advantage at all, and the entire data set needs to be scanned. Another way to tackle this problem is to create an index on a queried column, but we will demonstrate later that it is not the optimal solution.

In order to achieve better partition filtering, we introduce dynamic partition elimination (DPE). It is a collection of partition pruning strategies addressing various global hand data distribution patterns. Currently, we propose three different strategies (Direct, List, Fuzzy) to minimise the number of partitions while querying poker hand archives. They are supported by two different global indexing structures: the Player-Day Index (PDIX) and the To-Day Index (TDIX).

The PDIX implements the List pruning strategy and provides a complete list of partitions in which a given player was active. The index is split into a group of ordered files, each storing information about $k_f = 2^{16}$ players. In general, a PDIX_n file records the information about players with IDs from a range from $n2^{16}$ to $(n+1)2^{16}-1, n \in 0, 1, \ldots$. Individual files maintain an array of k_f slots that point to a list of partitions involving a player. To obtain the correct index file ID and the right slot number within a file, the following operations are required:

$$file\_id = \left\lfloor \frac{player\_key}{k_f} \right\rfloor, \quad slot\_id = player\_key \mod k_f \quad (7)$$

Player ID can be used to navigate within the index and retrieve the partition list (Fig. 7). The total storage cost of

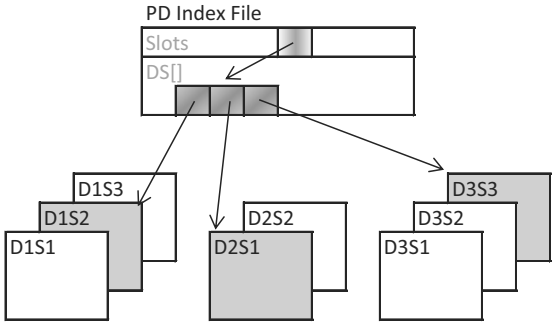

Fig. 7. Partition elimination based on the PD index.

the PDIX is

$$cost_{\text{PDIX}} = \left\lceil \frac{N_{pl}}{k_f} \right\rceil \left(k_f L_{sz} + \sum_{i=1}^{k_f} (e_{hdr} + |S_i|) \right), \quad (8)$$

where L_{sz} denotes the size of a single slot, e_{hdr} represents the size of the control field recording the number of the following segments and $|S_i|$ is the length of the array of partitions. Meanwhile, the cost of the TDIX is

$$cost_{\text{TDIX}} = I_{hdr} + N_s |e_{sz}|, \quad (9)$$

where I_{hdr} is the length of the TDIX header, N_s denotes the number of segments in the archive and e_{sz} is the length of a segment descriptor.

The remaining Direct and Fuzzy pruning strategies are supported by the TDIX. Compared with the PDIX, the TDIX is relatively small and inexpensive to maintain. It offers a direct, column to partition mapping when attributes such as $hand\_key$ or $start\_timestamp$ are queried. This is possible due to the fact that none of these attributes shares its value space with other partitions. In a situation where a range of attribute values overlap the adjacent partitions, the Fuzzy strategy is employed. Internally, the TDIX file is organised as an ordered list of partition descriptors. Each descriptor, called Segment Summary, records partition ID and the range boundaries of the supported attributes. The size of an individual segment summary item is only 47 bytes, whereas the total size of the index is around 530 KB. In order to find a search value in the list of segments, we use a modified version of a binary search algorithm, and in the worst case we find the value in $\log_2(N_s)$ moves, where N_s denotes the number of segments.

In addition, we use the Fuzzy strategy to obtain the list of partitions when hands for a particular tournament are searched for. Since the hands played in a single tournament can occupy many partitions, a single search is not sufficient. Thus, once a first matching segment is localised, we have to traverse the list in both directions to identify all other adjacent partitions that satisfy the search criteria.

Equipped with different pruning strategies together with predicate push-down (PPD) optimisation (Thusoo *et al.*, 2010), we can offer our version of an improved partition elimination algorithm. As a result of PPD optimisation, storage formats are exposed to expression trees containing a list of predicates specified by the user. By analysing the intercepted content of the predicate expression tree, we have a choice on which strategy to apply in order to minimise the search space. The predicate handler (PH) is responsible for extracting the predicates from the expression tree. Currently, it detects common SQL operators such as IN, BETWEEN, and the equality (EQ) and inequality (GT, LT) tests. In order to utilise the List pruning strategy, the PH also

decomposes complex predicates and allows replying to the expressions involving references to the built-in or user defined functions.

In addition, it detects whether only AND operators were specified, so that when multiple supported attributes are queried at once, the results obtained from many strategies can be combined together. For example, when searching for all poker hands that involve a specific player during a given tournament, two different pruning strategies are executed in response to PH tree analysis. Each strategy returns a sorted list of partitions. By applying the intersect algorithm, we can quickly produce the final set of partitions necessary to fulfil the query.

4.4. Hand data layout. Poker hand blobs along with hand describing attributes are stored row-wise in a file called segment. The size of an individual the segment is limited to 2 GB. In a situation where the daily hand volume exceeds the capacity of a single file, more segments are created. Consequently, a group of segment files can be produced for days with high poker activity. Every segment starts with a 64 B header, followed by the collection of hand rows. Figure 8 shows the layout a segment file and the format of the header. There

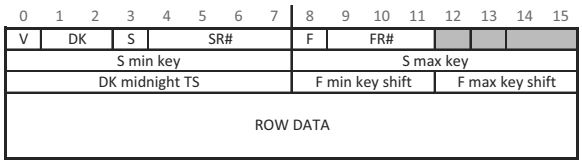

Fig. 8. Layout of the segment header.

are two one to each header. The first part is used to ensure the integrity of the entire segment and includes fields such as the version of a file, partition and segment ID, total number of rows, the number of fragments. In addition, to reduce the data-print size of the $hand\_key$ and $start\_timestamp$ columns, their minimum values are stored on the header with their deltas along the row. The second part is optional and contains the basic information about the fragment it belongs to. For example, it includes fields such as the number of rows stored in a fragment, the fragment's ID, etc.

Optionally, segments can be logically subdivided to form a complex structure comprising many adjoining fragments. When segment fragmenting is required, the header prepends every fragment. The size of a fragment can be chosen arbitrarily; in practice, however, it is set to the multiple of an HDFS block size. One disadvantage of fragmenting is that once the size of the fragment is specified it remains unchanged for the entire data set. In order to modify the size of all fragments, complete segment reorganisation is required.

The structure of a single hand row is presented in Fig. 9. The first three bytes of each row consists of two control fields tracking row length (2B) and the number of items in the field containing the array of player keys (1B). They are followed by a list of fixed size columns (33 bytes) and the array of players. The remaining bytes of every hand row are occupied by the poker hand blob.

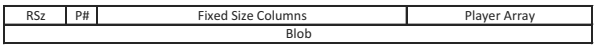

Fig. 9. Format of a single segment row.

Finally, the total storage cost of all segments can be estimated using the following formula:

$$cost_{seg} = \sum_{i=1}^{N_s} \Big(N_{f_i} S_{hdr} + N_{h_i} (r_{hdr} + |f| + 4\overline{N}_{pl} + \overline{b}) \Big), \tag{10}$$

where the symbols N_s and N_f denote the number of segments and fragments, respectively. The length of the segment and row headers is represented by S_{hdr} and r_{hdr}, while $|f|$ denotes the length of the fixed size columns and N_h represents the number of hands in a segment.

4.5. Offset-based segment indexing. Although partition elimination allows reducing the number of files to be processed, it only operates at a global, file level. In order to find all matching hand blobs within a single file, we need to employ additional indexing solutions. We achieve this by adding a support for the offset-based indexes. This offset is a relative position of a poker hand row from the beginning of a segment file. Since the size of an individual segment file is limited to 2 GB, a single offset value requires only four bytes when stored.

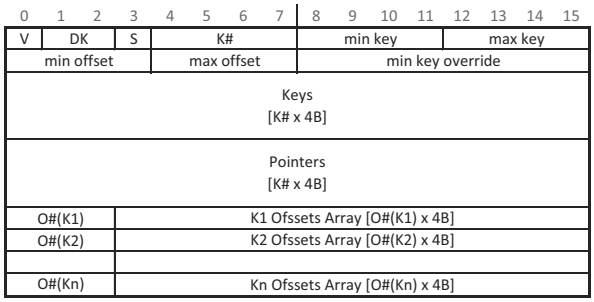

Fig. 10. Layout of the RecordReader Index (RRIX).

Our offset-based indexes (RRIXs) are comprised of four distinctive parts (Fig. 10). Each index file starts with a 64 bytes long header recording the number of stored keys n_k, the minimum and maximum value of the keys

and some additional fields ensuring its integrity. The index header is followed by two arrays containing n_k items. The first array maintains a sorted list of the indexed keys, while the second at a respective position, provides a pointer to the sorted list of offsets. Access to the index is achieved through a dedicated component called Index Reader. For a given set of keys, it returns sorted list of corresponding offsets.

Similarly to partitioning elimination, access to the expression tree is also possible at the record reader level. Once the list of segments is prepared, each of them is handled by an instance of a RecordReader class and executed by a dedicated computational container (governed by YARN). Before a reader starts accessing the data in a segment file, it refers to Predicate Handler to determine the existence of an expression tree. The handler validates if the content of the tree is viable to produce an optimised access path and prepares a list of accepted predicates. Next, using Index Reader, the PH coordinates the retrieval of the offset list for each predicate, before the final list is produced. At this stage the record reader is presented with a sorted list of offsets and commences a selective row read operation. Otherwise, when the predicate information is not available, it has to process the entire segment file.

Figure 11 shows how the information stored in the RRIX is used to produce a list of offsets to the hand rows. For a given key k, Index Reader uses the index header information to evaluate whether it belongs to a range from $\min\_k$ to $\max\_k$. If it does not, it returns an empty array indicating that the key is not present in a segment file and the reader finishes its work. When the key belongs to the range, Index Reader tries to find its position in the array of keys using the binary algorithm. Again, when the search is unsuccessful, an empty array is returned and the segment is not processed. However, a successful search returns the position of the key in the array. The same position value is used against the pointers array to retrieve the location of the list of offsets within the index file. Finally, the obtained list of offsets is returned to the reader.

The size of a single RRIX file can calculated using the following formula:

$$cost_{\text{RRIX}} = I_{hdr} + N_k 2|K| + \sum_{i=1}^{N_k}(c_{hdr} + |o_i|), \quad (11)$$

where I_{hdr} denotes header size and $|K|$ represents the length of the array of keys. In addition, c_{hdr} represents the size of the control field that maintains the number of the key's offsets and $|o_i|$ denotes the length of the i-th offset array. The RRIX structure can be also used to create a unique index. In this case, Eqn. (11) is reduced to the following form:

$$cost_{\text{RRUQ}} = I_{hdr} + N_k(|K| + o_{sz}), \quad (12)$$

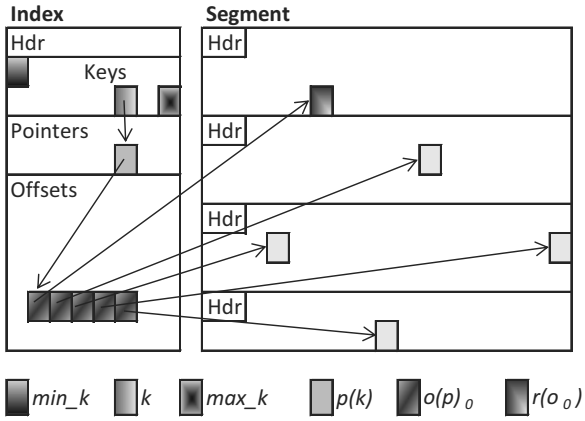

Fig. 11. RecordReader Index (RRIX) produces a list of offsets to hand rows.

where o_{sz} denotes the size of a single offset (four bytes).

4.6. Segment logical splitting. Segment files can be logically divided into fixed size fragments, typically 128 MB or 256 MB each. In a situation where a small number of segments needs to be accessed, splitting them into fragments allows better workload distribution between YARN containers. For example, if we need to process four segment files on a cluster that can allocate 100 processing containers, it is better to divide these segments into fragments and present them to record readers as a group of smaller discrete splits. Assuming 256 MB fragments and 2 GB segments, we would require 32 containers to complete the query. Thus, by splitting the workload into fragments we can potentially achieve an eightfold speedup over the undivided approach. In addition, dividing segments into fragments does not require changing the underlying structure of RRIXs. However, since the RRIX applies to the entire segment, additional offset values adjustment has to be performed by the record reader in order to align the fragment offset boundaries with the list of offsets returned by Index Reader.

4.7. Cost-based workload balancing. Apart from segment fragmenting, an additional performance increase can be achieved through a cost-based workload balancing. It provides a number of different optimisation techniques focused on minimising the costs associated with the retrieval of hand rows from a segment. It involves basic analysis of the offset list returned by Index Reader to Segment Reader. Factors such as the number of offsets, their density and distribution are used to prepare an improved row retrieval scheme. Currently, the number of offsets makes the biggest impact on the decision which scheme should be employed. In a situation where only a few offsets are processed, seek operations can offer better retrieval performance over sequential scans. On the other

hand, when a large number of offsets is considered, the SR can choose to perform multiple range scans instead of many expensive seeks. However, combining seek and scan operations together is not supported by any of the schemes.

4.8. Incremental load. Online poker is a 24/7 continuous operation focused on delivering the best possible gaming experience to a global player base. Such a complex and distributed software system practically eliminates the existence of any time frames with low or no poker activity. In our case, lack of a dedicated time window as well as the endless stream of poker hands requires a dedicated data loading solution.

One way to incrementally load poker hand stream to Hadoop is to append the hand data to the latest segment file of a relevant daily partition. Once the segment reaches the predefined size limit, it is closed and a new segment is created and starts receiving data from the stream. Since some of the hands take more time to finish than others and because multiple parallel systems are involved in their delivery, the cumulative stream of hand blobs cannot assure any logical ordering. Therefore, once a group of segments receives all the hands for a particular day, their content needs to be sorted in order to guarantee organisation of the stored blobs according to hand_key. Until the physical order of blobs across daily segments is restored, their RRIXs cannot be created. Furthermore, the absence of up-to-date RRIX causes an additional delay in updating both of the global indexes. Although this approach allows near real-time poker hand storage, it restricts the usage of our indexing structures in a situation where the most recent segments need to be processed.

Alternatively, by introducing narrow operational time windows, we are able to perform various maintenance tasks that would only be possible at the end of the day. This approach provides an opportunity to sort recent segment files and allows necessary index updates. The size of the window can be relatively small, 10 to 15 minutes, to guarantee minimal disruptions to hand querying services. Once the latest batch of hand blobs is processed, the updated segment files along with the indexes are transferred to the HDFS and exposed to Hadoop applications. Although we reduce the availability of real-time poker hand stream data to the end user, we guarantee a predictable data retrieval optimisation model which is consistent across the entire hand archive.

5. Data platform evaluation

In order to conduct a range of performance tests we use a physical 20-node Hadoop cluster. Its configuration includes 16 DataNodes, 2 NameNode and 2 Application Nodes. Each data node has a single 2.4 GHz Quad Core processor, 24 GB of the main memory and 16 $\times$ 500 GB SATA drives (2 for the OS, 14 for the storage). The name and application nodes have a 3 GHz Quad Core processor as well as 32 GB and 64 GB, respectively. All nodes are installed in 3 server racks and connected using 1 GB network adapters.

The tests we conducted to evaluate our framework were performed on an entire data set consisting of 12,000 segments containing 32 billion real-money hands out of all 40 billion. Due to the limited storage capacity of the production cluster, we only duplicated 10% of the tested data set size to convert it to ORC. The main goal of the conducted experiments was to determine the query execution performance and the impact on the cluster resources (IO and CPU). We prepared a set of queries that

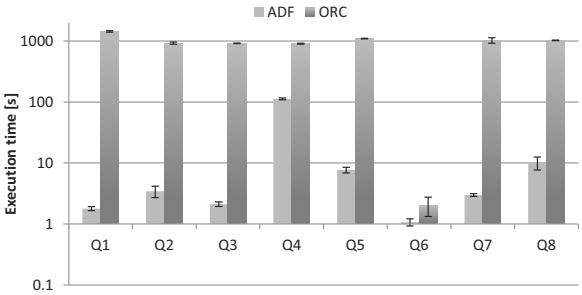

Fig. 12. Query performance comparison between the ADF and ORC.

reflect the most common scenarios when a hand table is used to

Q1: identify all the hands that occurred within a narrow time window, e.g., 1 minute (5425),

Q2: retrieve hands for a particular tournament (85),

Q3: find all hands involving a player that sat at a particular table in a given tournament (49),

Q4: obtain all the hands that completed on a specific table (56),

Q5: retrieve all the hands that involve a chosen player (35212),

Q6: identify a hand row by its ID (1),

Q7: search for all hands that were completed on one table in a selected tournament (49),

Q8: retrieve the last 10 hands played in a particular tournament (3218).

The values in the parentheses refer to the number of rows returned by the individual queries.

The results, obtained for both solutions and measured for all of the defined queries, are presented in Figs. 12 and 13. Each of the selected queries was executed 10 times

in order to determine their average run-times. Because of the significant performance differences between the tested solutions, the results are presented using the logarithmic scale. Figure 12 shows the average execution times, while Fig. 13 presents the number of splits accessed during the query execution. In addition, Fig. 12 includes the confidence interval of the population means. Using the value 0.05, we calculate a 95% confidence interval. It can be observed that the ADF offers a considerable advantage over the ORC file format in terms of speed and the number of accessed splits. It improves query execution times by more then 100 times over the ORC. While the ORC-based table gives access only to 10% of the entire set, it can be argued that the actual performance increase is thousandfold. This substantial performance increase is possible thanks to the extensive use of partition elimination and segment indexing. Queries that benefit from the global indexing structures complete within 5 to 20 seconds. Nevertheless, queries that cannot utilise global indexes (Q4) can still exploit segment indexes and complete up to 10 times faster than the ORC. In addition, we can also note that in a situation where multiple predicates are specified (Q3, Q7), increased performance is also achieved. Interestingly, both solutions achieve the same performance for query Q6. We expect that the reason for this behaviour lies in the fact that the $hand\_key$ attribute is defined as a first column in the hand table.

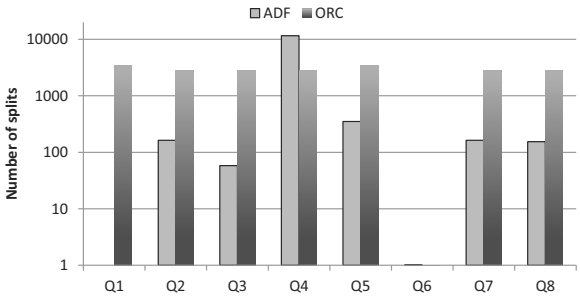

Fig. 13. Comparison of the number of accessed splits.

In addition, we present performance results of a legacy application used to generate poker hand transcripts. Hand transcripts are a complete textual representation of a hand object and all its associated events and attributes. They are frequently requested by customer service or internal audit teams. In practice, transcript generation is a very intensive operation. It involves finding, decompressing and deserialising all the relevant blobs in the archive.

We compare the results measured on the original RDBMS cluster hosting 60 nodes with the new version of the application running on Hadoop. Figure 14 shows that Hadoop not only provides better computational capabilities, but it also requires less hardware resources to compete with traditional databases.

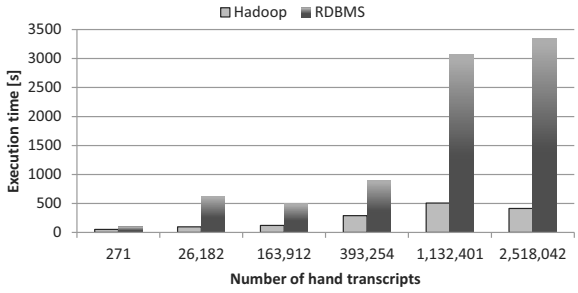

Fig. 14. Poker hand transcript generation performance comparison.

6. Conclusion

The ADF-based solution is deployed in a production environment which contains 40 billion hands. This system provides the business with unparallelled performance for both hand retrieval and complex data mining and analyses. The framework, in conjunction with the capabilities provided by Hive, allows users to take advantage of the parallel processing capabilities provided by Hadoop and MapReduce using a simple SQL-based interface.

The end user is abstracted from the underlying details and can focus on the retrieval or analysis task at hand. Fast identification and retrieval of relevant hand data pertaining to a given query, combined with the ability to do complex parsing, processing and filtering on the hand blob, mean that the user can do complex ad-hoc analysis in SQL. Legacy systems have had to identify and retrieve the data from potentially disparate sources and write custom, potentially throw away, code to do specific analyses.

The approach of systematically and iteratively pruning the search space for the data being queried has proven to be an excellent way to enhance the capabilities provided by Hadoop. Hadoop uses parallelism to perform at a huge scale. The ADF allows us the reduce, potentially drastically, the amount of data that need to be considered for a given query which, for the purposes of this paper, is poker hands.

The data are partitioned and indexed according to criteria that make sense for the given application, and this allows us to target the relevant files that will contain the query results. Even the files that contain the data themselves are indexed to provide for rapid retrieval of the relevant rows. This partitioning and multi-layered indexing forms a basis for the ADF, and it has delivered massive improvements in the poker hand domain. However, its applicability is general, so it could be employed in any other domains with similar properties.

We demonstrated that through careful storage design we can deliver better storage utilisation and improved performance over the traditional RDBMS. In addition, we showed that our storage model not only matches the

performance of the ORC file format, but in situations where complex structures are queried or high selectivity queries have to be supported, it offers an unrivalled alternative.

7. Future work

In the presented paper we described an efficient method of storing and retrieving poker hand objects in the Hadoop environment. Nonetheless, we also consider additional storage improvement techniques to further reduce the hand blobs data footprint. For instance, if we eliminate the need to keep the hand properties and the list of players along the blob attribute, we can save on the average 55 bytes per stored hand object.

One way to accomplish this would be to utilise an immense computational power of GPGPUs to decompress a hand blob to extract required attributes and make them available on demand. By excluding the poker hand auxiliary attributes from the physical store, we are not required to change the definition of the table storing hands in Hive, for the access to the data is achieved through Hive's SerDe[2] component, which separates the physical data layer from the logical table definition.

In terms of poker hand retrieval, we would like to enrich Hive's query language by extending its grammar so that poker-related statements could be easier formulated. The development of such a feature would offer increased expressiveness of a query language and provide an interesting avenue of research to pursue. For example, the use of the existing iterators could be replaced with intricate and poker-focused syntax structures.

Acknowledgment

This work was supported by statutory research funds of the Institute of Informatics, Silesian University of Technology, Gliwice, Poland (grant no. BK-230/RAu2/2017).

References

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A. and Rasin, A. (2009). HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads, *Proceedings of the VLDB Endowment* **2**(1): 922–933, DOI: 10.14778/1687627.1687731.

Alamoudi, A., Grover, R., Carey, M.J. and Borkar, V. (2015). External data access and indexing in AsterixDB, *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, Melbourne, Australia*, pp. 3–12, DOI: 10.1145/2806416.2806428.

Ambekar, G., Chikane, T., Sheth, S., Sable, A. and Ghag, K. (2015). Anticipation of winning probability in poker using data mining, *International Conference on Computer, Communication and Control, Indore, India*, pp. 1–6, DOI: 10.1109/IC4.2015.7375593.

Delaney, K. (2009). *Microsoft SQL Server 2008 Internals*, Microsoft Press, Redmond, WA.

Hadoop (2014). Apache Hadoop, `http://hadoop.apache.org`.

HDFS (2016). HDFS architecture, `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`.

Hive (2014). Apache Hive, `http://hive.apache.org`.

Jiang, D., Ooi, B.C., Shi, L. and Wu, S. (2010). The performance of MapReduce: And in-depth study, *Proceedings of the VLDB Endowment* **3**(1–2): 472–483, DOI: 10.14778/1920841.1920903.

Mealing, R. and Shapiro, J. (2015). Opponent modelling by expectation-maximisation and sequence prediction in simplified poker, *IEEE Transactions on Computational Intelligence and AI in Games* **PP**(99): 472–483, DOI:10.1109/TCIAIG.2015.2491611.

Miltersen, P.B. and Sørensen, T.B. (2007). A near-optimal, *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, Honolulu, HI, USA*, pp. 1168–1175, DOI:10.1145/1329125.1329357.

Mullins, C.S. (2000). *DB2 Developer's Guide, Fourth Edition*, Sams, Indianapolis, IN.

MySQL (2016). MySQL internals manual: Writing a custom storage engine, `http://dev.mysql.com/doc/internals/en/custom-engine.html`

ORC (2016). Apache ORC, `http://orc.apache.org/docs`.

PostgreSQL (2016). PostgreSQL documentation: Database page layout, `https://www.postgresql.org/docs/9.1/static/storage-page-layout.html`.

RCFile (2016). Apache Hive, `http://hive.apache.org/javadocs/r2.2.0/api/org/apache/hadoop/hive/ql/io/RCFile.html`.

Richter, S., Quiané-Ruiz, J., Schuh, S. and Dittrich, J. (2014). Towards zero-overhead static and adaptive indexing in Hadoop, *The VLDB Journal* **23**(3): 469–494, DOI: 10.1007/s00778-103-0332-z.

Shvachko, K., Kuang, H., Radia, S. and Chansler, R. (2010). The Hadoop distributed file system, *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, DOI: 10.1109/MSST.2010.5496972.

Teófilo, L.F. and Reis, L.P. (2011). Identifying player's strategies in no limit Texas Hold'em poker through the analysis of individual moves, *EPIA Conference on Artificial Intelligence, Lisbon, Portugal*, pp. 70–83.

Teófilo, L.F., Reis, L.P. and Cardoso, H.L. (2013). Estimating the probability of winning for Texas Hold'em poker agents, *IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Washington, DC, USA*, pp. 369–374, DOI: 10.1109/WI-IAT.2013.134.

[2]SerDe: Serialiser/Deserialiser (Hive, 2014).

Teófilo, Reis, L.P. and Cardoso, H.L. (2014). A profitable online no-limit poker playing agent, *Proceedings of the 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Washington, DC, USA*, Vol. 03, pp. 286–293, DOI: 10.1109/WI-IAT.2014.179.

Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H. and Murthy, R. (2010). Hive—a petabyte scale data warehouse using Hadoop, *Data Engineering (ICDE), 2010 IEEE 26th International Conference on, Long Beach, CA, USA*, pp. 996–1005, DOI: 10.1109/ICDE.2010.5447738.

YARN (2016). Apache Hadoop YARN, `http://hadoop.apache.org/docs/stable2/hadoop-yarn/hadoop-yarn-site/YARN.html`.

Marcin Gorawski is a professor at the Silesian University of Technology, Gliwice, Poland. His research interests include data and algorithms spaces theory, database systems, data warehouses, parallel programming, data stream processing, data privacy, real-time software engineering. M. Gorawski received his MSc, PhD and DSc degrees in computer science/informatics from the Silesian University of Technology, in 1978, 1986, and 2010, respectively. He is the author and a co-author of 350 journal articles, book chapters, and conference papers. He has also participated in almost 70 research projects/grants as well as over 200 IT industrial systems implementation and deployment.

Michal Lorek holds BSc and MSc degrees in computer science from the Silesian University of Technology, Poland. He specialises in databases systems and computer networks. Currently, he is a PhD student at the Silesian University of Technology. His research interests are focused on the real-time stream processing, data architecture and CUDA-powered big data solutions.

Received: 7 December 2016
Revised: 7 April 2017
Re-revised: 17 May 2017
Accepted: 5 June 2017