FlashQueryFile: Flash-Optimized Layout and Algorithms for Interactive Ad Hoc SQL on Big Data

Rini T Kaushik[1]

[1]IBM Research - Almaden

Abstract

High performance storage layer is vital for allowing interactive ad hoc SQL analytics (OLAP style) over Big Data. The paper makes a case for leveraging flash in the Big Data stack to speed up queries. State-of-the-art Big Data layouts and algorithms are optimized for hard disks (i.e., sequential access is emphasized over random access) and result in suboptimal performance on flash given its drastically different performance characteristics. While existing columnar and row-columnar layouts are able to reduce disk IO compared to row-based layouts, they still end up reading significant columnar data irrelevant to the query as they only employ coarse-grained, intra-columnar data skipping which doesn't work across all queries. FlashQueryFile's specialized columnar data layouts, selection, and projection algorithms fully exploit fast random accesses and high internal I/O parallelism of flash to allow fast and I/O-efficient query processing and fine-grained, intra-columnar data skipping to minimize data read per query. FlashQueryFile results in 11X-100X TPC-H query speedup and 38%-99.08% reduction in data read compared to flash-based HDD-optimized row-columnar data layout and its associated algorithms.

I. Introduction

Big Data warehouses such as Hive, BigQuery, BigSQL, Impala, and HAWQ are becoming common-place. Ability to interactively run ad hoc analytical OLAP queries over petabytes of data is becoming extremely important for businesses [12]. Traditional OLAP techniques for speeding up analytical queries such as pre-computed cubes don't work in the case of Big Data; dimensions in Big Data sets are many-fold and maintaining a cube across all dimensions is prohibitively expensive. Techniques such as indexing or caching don't help either given the ad hoc nature and the sheer size of the historical and recent data required by the queries. Recently proposed solutions for interactive analytics may not feasible for all as they rely on extreme scale-out [12], or approximate query processing [1]. A high performance storage layer that allows fast selections, projections, and joins over in-situ data is *vital* for interactive processing of ad hoc queries.

The Big Data warehouses don't manage the raw storage themselves and rely on the underlying distributed file system to store the data for them; F1/BigQuery uses Colossus file system and SQL over Hadoop systems such as Hive and Impala use Hadoop distributed file system (HDFS). Serializers/deserializers (serdes) such as RCFile [6], ORCFile [13], and Parquet [14] are used to store, read and write table data to the file system. While the columnar and hybrid row-columnar data layouts and algorithms of these serdes reduce disk IO by limiting data reads only to the columns present in the query [15], they still read significant data irrelevant to the query as illustrated in Section I-B. In these layouts, each column is horizontally partitioned into multiple row groups, and each row group consists of a configurable number of rows. A coarse-grained, intra-columnar data skipping technique was introduced in ORCFile to skip processing of some row groups; this technique yields effective data reduction only in a few scenarios (e.g., when column is sorted and has high cardinality). There is a need to develop I/O-efficient layouts and algorithms that allow *fine-grained* data skipping to reduce the irrelevant data read per query.

A majority of recent work leverages in-memory processing for interactive analytics [1], [12] and mandates large memory footprints. Flash is less expensive and allows much large capacities than DRAM; a DRAM + flash hierarchy stands to achieve much higher performance/$ than a large DRAM-only solution. DRAM's volatility mandates an additional durable copy of the data in a non-volatile medium, resulting in significant data duplication (not that desirable with Big Data). Flash is non-volatile and obviates the need for data duplication; it can serve as the primary and only storage for the data. Flash offers a 40-1000X improvement in random accesses and allows much higher degree of internal I/O parallelism than disks [3]. We make a case for leveraging high performance, non-volatile, small footprint, and low power storage mediums such as flash in the Big Data stack to speedup ad hoc query processing.

Architectural decisions of the storage layer in Big Data warehouses are based on the fundamental performance characteristics of HDDs; random I/O is avoided because of high seek times inherent with HDDs and sequential access is favored as much as possible. Fine-grained data skipping is avoided as it involves frequent seeks which come with huge performance penality on HDDs. Flash has a totally different performance paradigm from HDDs: seek times are negligible and random reads are much faster than sequential reads. A single flash device allows channel-, package-, die-, and plane-level I/O parallelism and can process multiple I/O requests concurrently unlike HDDs. HDD-optimized file formats and data access algorithms stand to attain suboptimal performance/$ by being placed in flash; sequential bandwidth of flash is only 2-7X higher than disks. In order to extract optimal performance/$ with flash, the transition from sequentially accessed HDDs to randomly and parallely accessible storage mediums such as flash mandates a *reexamination* of fundamental design decisions.

Previous work on using flash in databases mostly focuses on update-heavy, transactional databases and addresses slower random write performance of flash [8], [10], exploits flash as a buffer pool extension [5], or uses flash to store transactional logs [9], etc. [4]. Limited work has been done for leveraging flash in analytical processing (OLAP) workloads [11], [16], and no known work exists for leveraging flash for read-only Big Data SQL analytics ([7] for Big Data analytics is orthogonal). Tsirogiannis et. al. utilize PAX format [2] as-is, don't leverage I/O parallelism possible with flash, and intra-columnar data skipping happens only in the projection phase while entire columnar data is read during selection phase.

A. Contributions

To the best of our knowledge, we are the first to propose flash-optimized data layouts and algorithms to speedup Big Data analytical SQL query processing. The specific contributions are:

- New columnar data layouts and algorithms that leverage high random access performance and high internal I/O parallelism and concurrency of flash to perform fine-grained, intra-columnar data skipping and to speed up query processing as shown in Figure 1.
- New specialized selection-optimized columnar data layout and algorithm to allow very fast in-situ predicate match and reduce data read during selection phase.
- Specialized projection-optimized columnar data layout and algorithm that leverages late materialization and fine-grained data skipping to allow very fast data projection.
- High-performance, flash-aware coding to derive optimal performance/$ from flash.

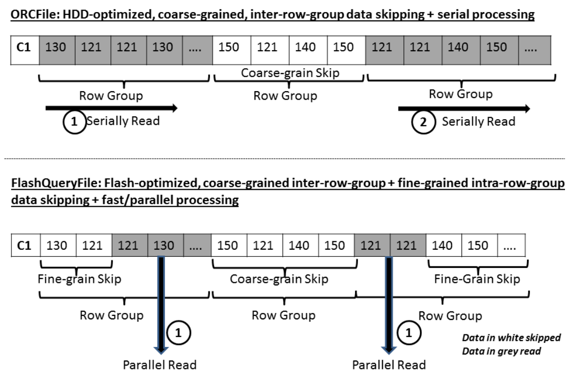

Fig. 1. Fine-Grained vs. coarse-grained data skipping

B. Opportunity for Intra-Columnar Data Skipping

To illustrate opportunities for data reduction via fine-grained data skipping in analytical queries, we consider TPC-H, an ad hoc decision support benchmark and we measure the data actually relevant to the queries by measuring selectivity (i.e., number of rows that match the predicate) of each selection column occurring in the TPC-H queries. As a motivating example, selectivity of each column in TPC-H Query 6 is shown in Figure 2. Only 15% of rows in selection column L_SHIPDATE, 18% in L_DISCOUNT, and 46% in L_QUANTITY match individual column query predicates. Only 1% of the rows match all the predicates across the three selection columns rendering only 1% data in projection columns L_EXTENDEDPRICE and L_DISCOUNT relevant to the query. Overall, only 16% data across the five columns is relevant to the query. Existing serdes may read as high as 99% of extra data from the two projection columns just to project 1% relevant data and as high as 50% extra data from the three selection columns. Similar trend exists in other queries and the selectivity of selection columns ranges from 1%-63% for a majority of TPC-H queries. Hence, an intra-row-group fine-grained data skipping technique that reads only the columnar data relevant to the query, in both the selection and projection phases, can lead to significant reduction in the data read per query.

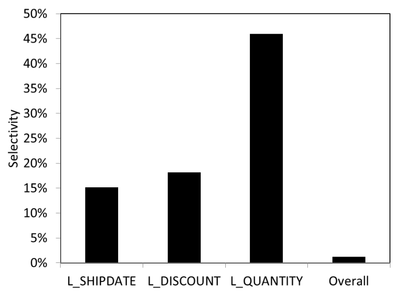

Fig. 2. Overall and per-column selectivity of each selection column in TPC-H query 6.

II. FlashQueryFile

In this section, we describe the data layouts, selection, and projection algorithms of FlashQueryFile.

A. Optimized Columnar Data Layouts

A table is logically represented as a FlashQueryFile file and consists of multiple distributed file system (DFS) blocks. Each block contains a block-level header (RCHeader) which contains sub-headers for each row-group (RGHeader) and each column (ColHeader) in the block. Rest of the block contains columnar data. Each RGHeader maintains offset to the start of the corresponding row-group in the block. Column headers contain offsets and synopsis (min, max, mid, and avg, count, sort order, cardinality, and field length) of the columnar data in the block. All the structures have been carefully selected with space-efficiency and performance in perspective. Instead of laying out columnar data exactly as ingested, FlashQueryFile lays out data by default in a hybrid selection- and projection-optimized manner to facilitate fast selection and projection. The choice of the layout is based on column characteristics (cardinality, popularity, sort order, etc.). The two layouts can be used independently as well. We first cover the selection part and then, the projection part of the layouts.

1) Selection-Optimized Data Layout: FlashQueryFile uses a selection optimized columnar data layout shown in Figure 3 to faciliate fast predicate match by quickly returning the set $R = [r_i, ..., r_j]$ of row ids that contain a given predicate value v, and to facilitate fine-grained data skipping. During the ingestion time, for each row group in the column, FlashQueryFile creates a dictionary of unique values from columnar data and lists of row ids where each unique value occurs in the row group. The dictionary is sorted and serialized contiguously on storage. Sorting is important for allowing fast predicate match and clustered reads of lists of row ids. Next, list of row ids and an offset to list is serialized for each unique value.

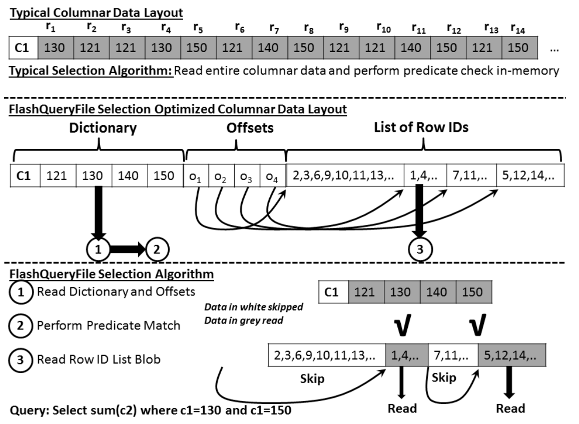
Typical Columnar Data Layout
Typical Selection Algorithm: Read entire columnar data and perform predicate check in-memory
FlashQueryFile Selection Optimized Columnar Data Layout
Dictionary
Offsets
List of Row IDs
FlashQueryFile Selection Algorithm
Read Dictionary and Offsets
Perform Predicate Match
Read Row ID List Blob
Data in white skipped
Data in grey read
Skip
Read
Query: Select sum(c2) where c1=130 and c1=150

Fig. 3. Selection optimized layout and algorithm

Storing columnar data as a dictionary of unique values and associated list of row ids is very space-efficient when size of the column data type is $>$ 4 bytes and when the cardinality is low/medium. Some exceptions to the layout: 1) if the cardinality of the column is 1 (i.e., all values are unique), there is no value in creating a dictionary and data is laid out as is, and 2) if the column is sorted, a more concise data structure is used that stores just the start and end row id for each unique value instead of storing all intermediate row ids.

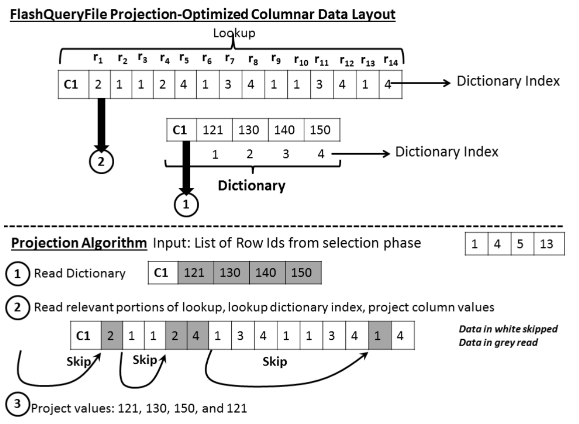
FlashQueryFile Projection-Optimized Columnar Data Layout
Lookup
Dictionary Index
Dictionary
Projection Algorithm Input: List of Row Ids from selection phase
Read Dictionary
Read relevant portions of lookup, lookup dictionary index, project column values
Data in white skipped
Data in grey read
Skip
Project values: 121, 130, 150, and 121

Fig. 4. Projection optimized layout and algorithm

2) Projection-Optimized Data Layout: The projection optimized data layout as shown in Figure 4, is designed to quickly read column value v stored at given a row id r_i. The layout has three flavors based on column cardinality, sort order, and data type (fixed vs. variable length). Space-efficiency and performance considerations play a role in the layout decision. If the column cardinality is small/medium, a lookup structure is created to map each row id to the dictionary (described in selection layout) index of its column value; the space overhead of the lookup structure gets fully masked by the space savings achieved by storing data in a dictionary format.

If column cardinality is $=$ 1, no lookup structure is used. If column is sorted, an optimized lookup structure is used that maps ranges of row ids as opposed to individual row ids. If column cardinality is very high and column is not sorted, no dictionary layout is used in a projection-only layout; data is simply laid out sequentially in interest of performance. While, the simple layout is not as space-efficient as the dictionary layout, it yields higher performance as it eliminates a large dictionary read and enables clustered read of row id values which leads to better performance. If column is variable length, a lookup structure is maintained that maps each row id in the column to the offset of its column value in the columnar data. If column is fixed length, no lookup structure is used and offset of column value of interest is simply calculated using specified row id and field length. Column header maintains the offset of the lookup structure and the dictionary and/or data.

B. Flash-Optimized Algorithms

FlashQueryFile's algorithms are designed to fully leverage flash's high performance random accesses and high internal IO parallelism. The algorithms are coded using high-performance apis/methods, a mandatory feat for leveraging performance from flash. Low-level NIO file channel api, and non blocking, direct, and asynchronous IO was used instead of high-level buffered file streams to prevent unnecessary data prefetches, data copies, and to randomly access data at desired offset with minimal overhead. Instead of using inefficient inbuilt Java class serialization and deserialization, highly efficient data marshalling is used. The selection and projection algorithms are described in detail below.

1) Flash-Optimized Selection Algorithm: The selection algorithm described in Algorithm 5 is designed for fast predicate match and coarse- and fine-grained data skipping. Steps 2-8 perform coarse-grained inter-row group skipping for all selection columns by examining summary information in the column subheaders stored in RCHeader. All the row groups whose min and max values are out-of-range compared to query predicates are skipped.

Steps 11-32, describe the fine-grained data skipping part of the algorithm. FlashQueryFile spawns threads in parallel to process data in each of the remaining row groups; parallelism is enabled by high internal I/O parallelism allowed in flash. The desired portion of the dictionaries of the selection columns are deserialized from respective file offsets u_offset or u_mid_offset specified in the column headers. The size of dictionary read is much smaller than the actual columnar data if the cardinality of the column is low/medium, or else, if only a portion of the dictionary is relevant to the predicates.

A predicate match is performed on the unique values contained in the dictionary. The predicate match considers the cardinality of the selection columns in its column processing order as that has a direct impact on the data reduction. If predicate match fails, none of the row id blobs are read for the column and processing of the entire row group is skipped. Such skips significantly reduce the amount of data read by the query. As an illustration, consider column L_QUANTITY, one of the selection columns in TPC-H query 6. Its cardinality is 0.0008%; resulting in a dictionary with 83 unique values ($<$ 500

```
Require: Column predicates in query
 1: return Set of row ids that match the all the query predicates
 2: Fetch the block header RCHeader from the FlashQueryFile block. Cache it for future queries
 3: for all Row groups in the FlashQueryFile block do
 4:     for all Selection columns do
 5:         Compare the predicate with the min and max value stored RCHeader− >RGHeader− >ColHeader
 6:         if predicate is smaller than min or greater than the max then
 7:             Skip the row group from processing completely by marking it skipped
 8:         end if
 9:     end for
10: end for
11: for all Remaining row groups that needs to be processed do
12:     Spawn a thread to process the row group
13:     Order selection columns based on cardinality
14:     for all Selection columns in query do
15:         if Predicate value $>$ mid value of column then
16:             Retrieve column's dictionary from file block offset ($u_{mid}\_offset$)
17:         else
18:             Retrieve column's dictionary from file block offset ($u\_offset$)
19:         end if
20:         Match predicate value with unique values in column's dictionary
21:     end for
22:     if Predicate matches in all selection columns then
23:         for all Selection columns in query do
24:             Read offsets of row ids list blobs from file block
25:             for all Unique values that match predicate do
26:                 if Unique values are consecutive then
27:                     Coalesce multiple row id list blobs and read them in one call
28:                 else
29:                     Read each individual row ids list blob from file block
30:                 end if
31:             end for
32:         end for
33:     else
34:         Stop processing the entire row group and rest of the columns
35:     end if
36:     Find sorted intersection of row ids that satisfy all the predicates
37: end for
```

Fig. 5. Flash-Optimized Selection Algorithm

```
Require: Set of row ids that match all the query predicates
 1: return Projection of the desired attributes from database matching the predicates
 2: for all Projection columns do
 3:     for all Row groups with non-empty set of final row ids do
 4:         if Projection column is laid out as-is then
 5:             if Row ids are not clustered together then
 6:                 Examine the column header and identify the offset of the lookup data ($l\_offset$) structure and offset of data ($l\_data$)
 7:                 Calculate index of desired row id in the lookup data structure
 8:                 Calculate file offset of the column value pertaining to the row id = $d\_offset$ + offset stored in lookup
 9:                 Seek to the offset of the column value in file block and read column value
10:             else
11:                 Cluster row ids into multiple projection buffer size chunks and read each chunk
12:             end if
13:         else
14:             Read in the dictionary
15:             if Row ids are not clustered together then
16:                 Calculate index of desired row id in the lookup data structure and read dictionary index stored there
17:                 Lookup dictionary value at the index
18:             else
19:                 Cluster row ids and read portions of lookup structure for each cluster
20:                 Figure out dictionary entry pertaining to each row id
21:             end if
22:         end if
23:     end for
24: end for
```

Fig. 6. Flash-Optimized Projection Algorithm

bytes in size) and associated list of row ids with approximately 12k row ids each for a row group of 10 million rows. If predicate match were to fail for L_QUANTITY, a minuscule read of $<$ 500 bytes leads to a skip of 83 lengthy lists of row ids. If predicate matches one or more unique values, the list of row ids of these values are read in parallel. Remaining lists of row ids are skipped, resulting in significant data reduction when the query selectivity and cardinality of the column is low (i.e., very few unique values match the predicate).

Finally, a sorted intersection of row ids that match all the predicates is passed to the projection algorithm discussed next. While the selection algorithm performs best on flash, it also performs well with data present on disk. Flash's advantage shows up at higher cardinality and in situations needing a lot of data skips and very small data reads.

2) Flash-Optimized Projection Algorithm: FlashQueryFile reads data from projection columns only after the selection phase is completed. Such late materialization allows FlashQueryFile to limit the data that needs to be read during the projection phase to only the relevant row ids. For example, in the illustrative query 6 in Section I-B, only 1% of rows will need to be read for each of the two projection columns, resulting in an impressive skip of of 99% data per column. In contrast, the existing serdes don't use late materialization and the data for the selection and projection columns is usually read upfront.

FlashQueryFile carefully considers the selectivity (i.e., size of set of final row ids that match all predicates) of the query in attaining a fine-balance between random and clustered calls. If the selectivity of the query is very low ($< 2\%$), FlashQueryFile processes each row id of interest individually in the file block and reads in column value; randomly accessible storage mediums such as flash are a key enabler for making such targeted random reads possible without a performance impact. While, the targeted approach allows reading just the relevant data, it also results in a separate file system call for each row id where each call has a operating system overhead associated with it. Hence, FlashQueryFile switches to a clustered read approach if the selectivity of the query is high to reduce the overhead of the file system calls. The dictionary is read in its entirety and the lookup elements of the row ids of interest are clustered and a projection buffer worth of data is read per file system call.

III. Evaluation

We used 40GB lineitem table (largest table in TPC-H benchmark) with 240+ million rows and 16 columns of varying lengths and characteristics. We compared the performance of FlashQueryFile serde with state-of-the-art ORCFile serde and placed both on flash. The evaluation was done on a single server of the IBM Research's Accelerated Discovery Lab cluster. The server had six quad-core Intel(R) Xeon(R) CPU E5645@ 2.40GHz, Fusion-IO 825GB ioScale2, and 96GB RAM. Red Hat Enterprise Linux Server release 6.5 (Santiago). Java(TM) SE Runtime Environment (build 1.7.0_40-b43). Each row group consisted of 10 million rows and 24 threads were used to process each row group in parallel. Figure 7 shows the speedup of query 1 with FlashQueryFile compared to ORCFile (also placed on flash). Speedup of 11X-103X was achieved compared to

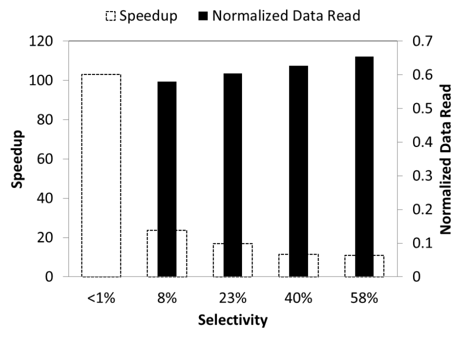

Fig. 7. Speedup and data read by TPC-H Query 1 with FlashQueryFile vs. ORCFile.

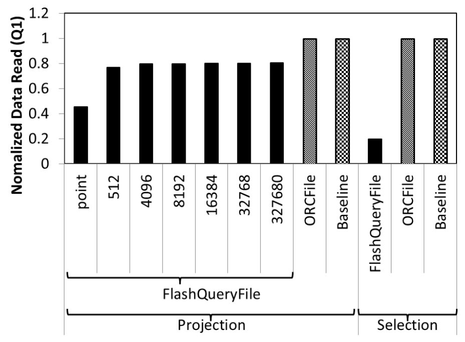

Fig. 8. Normalized data read in selection and projection phase of TPC-H Query 1 with selectivity of 40%.

ORCFile with selectivity ranging from <1% to 58%. Even 98% selectivity yields a speedup of 4X. Results reiterate the importance of using flash-optimized layouts/algorithms to leverage full performance/$ from flash. Simply placing an HDD-optimized layout on flash and accessing it using HDD-optimized algorithms results in suboptimal performance on flash.

FlashQueryFile reduces overall data read during query courtesy of its fine-grained data skipping and encoding as shown in Figure 7. Figure 8 illustrates data read in the selection and projection phase of TPC-H query 1 (with selectivity 40%) with FlashQueryFile normalized to baseline (e.g., a serde like RCFile which reads the columnar data in its entirety in both the selection and projection phases as it doesn't employ any data skipping logic). ORCFile only has a coarse-grained mechanism in place for filtering out row groups based on min and max summary information. The mechanism fails to find any row group to skip and ends up reading the entire data in the selection column for query 1. On the other hand, FlashQueryFile yields a reduction of 81% in the data from the selection column l_shipdate. Cardinality of l_shipdate is low; the dictionary size is small and each row id blob is large. The row id blobs that don't match the predicate get skipped, resulting in a data read of only 40% of row id blobs. Further reduction in data read is courtesy of space-efficient dictionary based layout which is able to mask other overheads introduced by the optimized layouts. Data reduction happens during the projection phase as well as FlashQueryFile uses late materialization and reads data only for the final set of row ids that match all the predicates. The data reduction depends on the projection buffer size; lower the size, higher is the data reduction. However, performance suffers at very small projection buffer sizes when selectivity is $> 1\%$ because of the overhead of the increased system calls.

IV. Conclusion

Storage performance plays a vital role in the performance of ad hoc analytics over Big Data. Existing columnar and row columnar file formats and associated table scan algorithms read lot more data than actually needed by the query and are optimized for HDD performance characteristics. Simple placement of HDD-optimized layouts on flash stands to derive only limited performance/$; flash-optimized file formats and algorithms are required to derive full benefit of flash. In this paper, we proposed a new flash-optimized data layout called FlashQueryFile and its associated selection, projection, and injection algorithms. Evaluation results with TPC-H dataset show that FlashQueryFile is able to significantly enhance query performance and throughput by: 1) reducing the data read during both selection and projection phases of query processing courtesy of its data structures and layouts that allow fine-grained, intra-row-group data skipping, 2) exploiting high internal IO parallelism and fast random accesses allowed by flash, and 3) using high performance flash-optimized code. Our results show that flash can be effectively leveraged in Big Data stack to speed up ad hoc OLAP style SQL queries.

References

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. EuroSys '13.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB '01*.

[3] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA '11*.

[4] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *SIGMOD '13*.

[5] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging dbms buffer pool using ssds. In *SIGMOD '11*.

[6] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE'11*.

[7] S.-W. Jun, M. Liu, K. E. Fleming, and Arvind. Scalable multi-access flash store for big data analytics. In *FPGA '14*.

[8] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD '07*.

[9] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD '08*.

[10] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, Sept. 2010.

[11] M. McCarthy and Z. He. Efficient updates for olap range queries on flash memory. *Comput. J.*, 54(11):1773–1789, Nov. 2011.

[12] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, June 2011.

[13] Orcfile. cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC.

[14] Parquet. http://parquet.io/.

[15] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB'05*.

[16] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD '09*.