Performance Evaluations of Document-Oriented Databases using GPU and Cache Structure

Shin Morishima
Dept. of ICS, Keio University,
3-14-1 Hiyoshi, Kohoku, Yokohama, Japan
Email: morisima@arc.ics.keio.ac.jp

Hiroki Matsutani
Dept. of ICS, Keio University,
3-14-1 Hiyoshi, Kohoku, Yokohama, Japan
Email: matutani@arc.ics.keio.ac.jp

Abstract

Document-oriented databases are popular databases, in which users can store their documents in a schema-less manner and perform search queries for them. They have been widely used for web applications that process a large collection of documents because of their high scalability and rich functions. One of major functions of document-oriented databases is a string search that requires a high computational cost for a large collection of documents, because its computational complexity increases as the documents increase. In document-oriented databases, a database index is typically used for improving text search queries. However, the index cannot always be used for text search queries, such as a regular expression match search. To accelerate such queries by using GPUs, in this paper, we propose a GPU-friendly cache structure, called DDB Cache (Document-oriented DataBase Cache), which is extracted from a document-oriented database. By using GPU and DDB Cache, we can improve a performance of text search queries without relying on the database indexes. We implemented DDB Cache for MongoDB. Experimental results using GeForce GTX 980 show that our approach improves the performance of regular expression search queries by up to 101x compared to the original document-oriented database.

1. Introduction

Structured storages or NOSQLs [1] are databases that store and retrieve information in a simpler and more flexible data structure compared to conventional RDBMSs (relational database management systems). Because of the simplicity of their data structure, certain types of operations are faster than RDBMSs. In addition, a horizontal scalability is often much enhanced by means of partitioning (sharding) and replication over a number of machines. They are increasingly used for Web applications and cloud-based systems that handle Big data.

Various NOSQLs have been developed and they are loosely classified into four categories: key-value store, column-oriented store, document-oriented database, and graph database. Because NOSQLs are typically optimized for some operations or purposes, one or some of them are used depending on applications. According to the DB-Engines Ranking [2] (as of April 2015), all the top three popular databases (e.g., MySQL) employ the relational model, while the fourth one is MongoDB which is a document-oriented NOSQL. Document-oriented databases [3][4] store data as JSON-like documents without a prior knowledge of keys and data types. They are sometimes referred as schema-less databases [1]. The stored documents can be retrieved based on the field, range, and/or regular expression pattern. Their horizontal scalability can be enhanced via partitioning and replication.

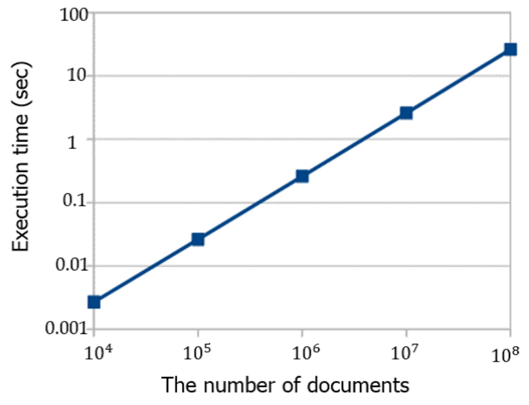

Figure 1. Text search execution time vs. the number of documents in document-oriented database

One of powerful functions of document-oriented databases is a text processing that supports regular expression based text search queries. A text processing time increases as the number of documents increases. Figure 1 shows the execution time of MongoDB to perform a text search when the number of documents (each document contains eight characters) is increased. Both X- and Y-axis scales are logarithmic. The result shows that the execution time is proportional to the number of documents and a single search query takes 26 seconds for 100 millions documents.

1. In practice, some sort of schema is expected when retrieving data efficiently.

Database indexing is a common technique to reduce the time complexity of an information retrieval from O(n) to O(log n). MongoDB supports database indexing so that users can build indexes of one or more fields for a collection of documents. These indexes are stored in a B+tree based data structure for a faster lookup compared to a full scan. However, such indexes cannot be used for all the queries. For example, regular expression based text search queries cannot utilize the index tree and a full scan is performed instead, because indexes are built with an assumption that words go from left to right. Instead of the B+tree based index, a q-gram based index [5] (in which a text document is broken into substrings of length q and these substrings are indexed) has been studied to support regular expression based text search queries [6][7]. However, such a q-gram based index consumes more memory space and takes more time to build the index at every write queries compared to the B+tree based index; thus it further degrades the database write performance (we will evaluate the write throughput degradation in Section 5).

One of notable features of document-oriented databases is related to their scheme-less data structure. The application performance is typically restricted by the slowest queries; thus, boosting their rich text search queries is highly required for a wide range of applications that process Big data. Recently, GPUs have been applied to various text processing, such as regular expression matching [8][9][10]. However, there is no prior work that addresses how to utilize GPUs for such emerging document-oriented databases. In this paper, we propose a simple Document DataBase (DDB) Cache structure suitable for a GPU-based text processing. We can accelerate the slow search queries that cannot utilize database indexes, by combining DDB Cache and GPU. DDB Cache is embedded into a document-oriented database (called a host database) and the cached data are updated by the host database. We demonstrate a double-digit performance improvement for regular expression based text search queries on MongoDB by introducing DDB Cache combined with GeForce GTX 980 GPU.

The rest of this paper is organized as follows. Section 2 surveys related work and Section 3 introduces the target document-oriented database. Section 4 proposes DDB Cache for GPUs. Section 5 evaluates our DDB Cache and the conventional index-based approaches in terms of performance. Section 6 concludes this paper.

2. Related Work

2.1. SQL and NOSQL Databases using GPUs

Key-value store, such as Memcached and Redis, is another class of structured storages that stores data as pairs of key and value. In [11], Memcached is accelerated by GPUs. The results show a significant speed-up respect to the original software implementation, while the performance gain is reduced when CPU-GPU data transfer overhead is considered.

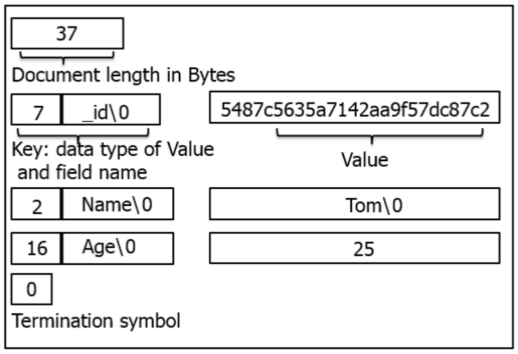

Figure 2. Example of document data structure expressed in BSON format

On the other hand, text search queries used in document-oriented databases are compute-bound (computation intensive) and their performance can be enhanced by utilizing the massive parallelism of GPUs. That is, such a performance gain is expected to be much larger than the CPU-GPU data transfer overhead. Similarly, graph search queries used in graph database are also compute-bound and their acceleration using GPUs is reported in [12].

In addition, SQL database operations are accelerated by using a GPU and CUDA in [13]. They introduce a GPU implementation of SQLite command processor.

2.2. Regular Expression Matching using GPUs

GPU-based acceleration methods for a regular expression matching that utilizes NFA (Non-deterministic Finite Automaton) or DFA (Deterministic Finite Automaton) have been studied so far. In [8], a regular expression matching is accelerated by GPUs using a DFA-based approach. In [9], a small rule-set based regular expression matching is accelerated by GPUs. An NFA-based approach is reported in [10]. The benefit of the NFA-based approach is a higher memory efficiency.

To show that our GPU-friendly DDB Cache mechanism can be used for a regular expression matching in document-oriented databases, we implement a simple DFA-based approach (similar to [8]) for regular expression based text search queries and evaluate it in Section 5.4.

3. Document-Oriented Databases

As a target document-oriented database, in this paper, we use MongoDB which is one of the famous open-source document-oriented databases implemented in C++ [3].

3.1. Data Structure of MongoDB

In MongoDB, documents are represented in a data interchange format called BSON, which is a binary form of JSON (JavaScript Object Notation). Figure 2 shows an example of a document that has three fields (i.e., _id, Name, and Age) expressed in BSON format. MongoDB maintains a vast number of documents, each of which consists of three parts: 1) a header that represents the document length in Bytes, 2) Key-Value pairs, and 3) a termination symbol that represents the end of a document. For each Key-Value pair, Key part consists of a field name and Value data type. Value part represents a field value in the data type specified in the Key part. MongoDB supports 19 data types, such as ID, UTF-8 String, 32-bit Integer, 64-bit Integer, Double, Boolean, Date, and Timestamp. In Figure 2, data types 7, 2, and 16 are corresponding to ID, UTF-8 String, and 32-bit Integer, respectively.

3.2. Queries in MongoDB

When a search query that specifies search conditions for some fields of documents is given to MongoDB, the corresponding documents that match to the conditions are retrieved. For example, when a search query that specifies a condition of "Name == Tom && Age == 25" is performed, the document illustrated in Figure 2 is returned, because it matches to the condition. Search conditions are not needed for all the fields. Actually, no condition is specified for _id field in this example. In addition to perfect-matching, various search conditions, such as less-than, greater-than, and range conditions for Integer types and pattern match conditions for String types, can be used.

The time complexity of a full scan for n documents is O(n), because a given search condition is examined for every document. Because such a linear scan is not acceptable especially for large data sets, database indexing is supported in MongoDB. One or more selected fields of documents are extracted, sorted in an ascending (or descending) order, and stored in a B+tree data structure. By traversing the B+tree based index, the time complexity to search for n documents is reduced to O(log n). However, document search queries that contain regular expression based conditions cannot exploit the B+tree based index, because the selected field-values are sorted from the leftmost characters. In this case, an O(n) full scan is performed instead. Thus, such queries become a performance bottleneck.

Recent MongoDB provides the text index for text search of string contents, in addition to a conventional B+tree based index. Although it still cannot handle regular expression based text search, it can search a specified word in the documents in a collection. However, as stated in the MongoDB documentation [14], building a text index is similar to building a large multi-key index and will consume more storage space and take a longer time compared to a conventional index; thus it will impact insertion throughput. In Section 5, we will show that even a conventional index, which is faster than the text index, significantly degrades the write performance.

In the next section, we propose DDB Cache mechanism in order to accelerate such queries by using GPUs.

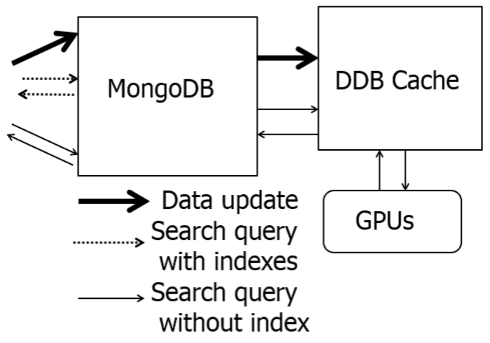

Figure 3. Proposed system using DDB Cache

4. DDB Cache Mechanism

DDB Cache is extracted from host databases and used for accelerating text search queries with GPUs.

4.1. Overall System Using DDB Cache

Figure 3 shows the proposed system that consists of a document-oriented database (i.e., MongoDB), the proposed DDB Cache, and GPUs mounted in the same machine. DDB Cache is created on the host memory by extracting necessary information from the host database. Because data structure of DDB Cache is designed for GPU-based text processing, the cached data can be directly transferred to a GPU memory. A text search is performed as follows, depending on the query types.

- For update queries that write a host database, they are processed by the database first and then DDB Cache is updated accordingly.
- For search queries that utilize database indexes, they are processed by the database without DDB Cache and GPUs. The search result is returned from the database.
- For search queries that cannot utilize database indexes, they are processed by GPUs with DDB Cache. The result is returned from GPU+DDB Cache via the database.

4.2. DDB Cache Creation

Figure 4 shows a data structure of DDB Cache. It also illustrates how to create DDB Cache by extracting data from MongoDB. The upper half of the figure shows a collection of simplified documents stored in BSON format and the lower half of the figure shows the corresponding DDB Cache structure. Again, MongoDB stores multiple documents, each of which consists of multiple fields, each of which is represented in a Key-Value pair. DDB Cache is created by extracting the selected fields of all the documents and then storing only their values into a linear array (called Value array) for each field. Because the

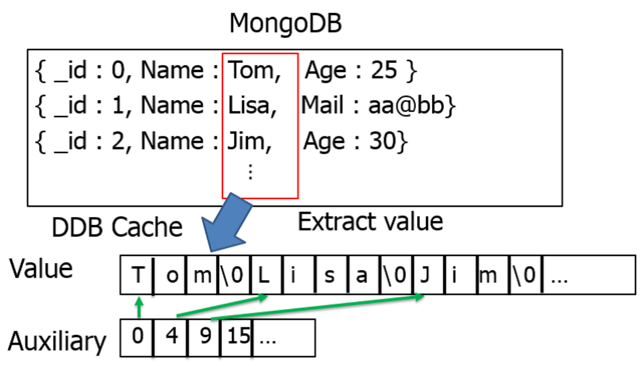
MongoDB
{ _id : 0, Name : Tom, Age : 25 }
{ _id : 1, Name : Lisa, Mail : aa@bb}
{ _id : 2, Name : Jim, Age : 30}
⋮
DDB Cache
Extract value
Value
T o m \0 L i s a \0 J i m \0 ...
Auxiliary
0 4 9 15 ...

Figure 4. DDB Cache creation

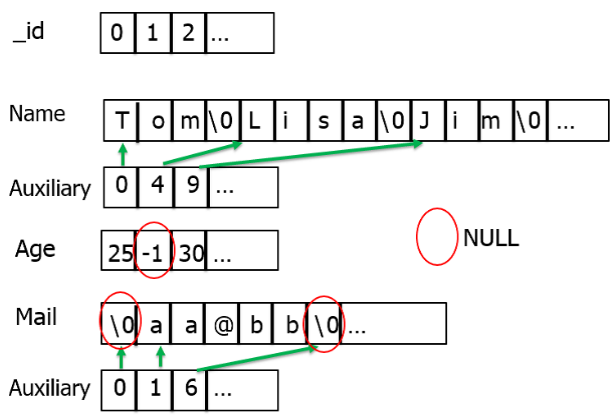

Figure 5. DDB Cache for schema-less structure

field name and data type of every document in the same linear array are the same, a pair of field name and data type is stored for each Value array (not needed for every document). That is, a memory footprint of these Keys can be reduced from $n \times (c+1)$ Bytes to $(c+1)$ Bytes, where n is the number of documents and c is the field name length. In Figure 4, for example, Tom, Lisa, and Jim are extracted from Name field of three documents and stored into a single Value array (labeled as Name).

However, assuming these strings (e.g., Tom, Lisa, and Jim) are simply stored into a Value array, a string search for such a linear Value array cannot be efficiently parallelized, because a sequential linear search is required to find a terminal symbol of each string before performing a parallel search. Thus we introduce Auxiliary array that maintains a start address of each string in the Value array. In this example, the Auxiliary array shows that start addresses of Tom, Lisa, and Jim are 0, 4, and 9 in the Value array, respectively. Auxiliary array is needed only for data fields in variable length data types, such as String, while it is not needed for those in fixed length data types, such as Integer. For search queries that specify a condition that span multiple fields (e.g., Name and Age), corresponding Value arrays are fed to GPUs for a parallel search.

4.3. DDB Cache Update

DDB Cache can be dynamically updated in response to write queries on the host database. In addition, because document-oriented databases are intuitively schema-less and valid fields may differ between documents (e.g., a document has fields A and B, but another document has field C only), DDB Cache must cope with such cases.

Although documents may not have values for all the fields, all the documents must have _id field as a primary key. Value array of _id field is thus created as a primary key in DDB Cache too. Value arrays of the selected fields, which are expected to be used in search queries, are then created for DDB Cache.

Figure 5 illustrates DDB Cache applicable for such schema-less cases. The contents are the same as those in Figure 4. In the DDB Cache, there are three Value arrays for _id, Name, Age, and Mail fields and two Auxiliary arrays for Name and Mail fields, because these two fields store variable length strings. Since every document has _id field, the number of values stored in _id Value array represents the number of documents in the database. For fixed-length data types (e.g., Integer), the number of values stored in its Value array is also equal to the number of documents. If a document has a valid value for the field, the value is stored in the Value array; otherwise, a termination symbol that represents NULL is stored instead. For variable-length data types (e.g., String), both Value array and Auxiliary array are created for each field. The number of values stored in the Auxiliary array is equal to the number of documents, because each value in the Auxiliary array represents a start address of the corresponding string in the Value array. If a document does not have a valid string value, only a termination symbol that represents NULL is stored in the Value array, so that search queries simply skip such documents. In Figure 5, -1 is stored in Age Value array if a document does not have Age value, and only a termination symbol is stored in Mail Value array if a document does not have Mail value [2]. Field values of a document can be directly read from Value arrays for fixed-length data types or indirectly read via the corresponding Auxiliary arrays for variable-length data types. When we look at the third document in Figure 5, values of _id, Name, Age, and Mail represent 2, Jim, 30, and NULL, respectively.

They are consistent with the third document in the host database, as shown in Figure 4.

In case new documents are added or existing documents are elongated (e.g., when a shorter string is replaced with a longer one) in the host database, a certain amount of margin is preallocated at the tail of each Value array in DDB Cache. Such unused regions are filled with a termination symbol beforehand and can be used for newly-added or

2. If a termination symbol cannot be predefined for some fields (e.g., -1 may be stored in Integer fields as a valid value), we simply use Auxiliary array for such fields, in order to identify empty fields. Alternatively, an additional bitmap, in which 0 indicates empty and 1 indicates valid for each field, can be used as a more efficient structure.

elongated documents afterward. When an update operation replaces a string value with a longer one, the longer one is stored in the margin region and the corresponding Auxiliary array is updated to point the new address stored in the margin region. If a margin region in a Value array is running out, a larger memory is reallocated for the Value array with more margin. Please note that we do not have to distinguish empty fields from a margin region in DDB Cache, because both empty and margin fields cannot be matched to specified search conditions; thus a margin region does not affect the search results.

To exploit GPUs with DDB Cache, slight modifications are needed for the host document-oriented database. When a document in the host database is updated, the corresponding values in DDB Cache must be searched and updated accordingly. To eliminate such a search cost in DDB Cache, a lookup table that keeps track of the index number of each document stored in DDB Cache is added to MongoDB. Using this lookup table, the search cost at MongoDB when DDB Cache is updated becomes O(1).

Assume an update operation of MongoDB that adds a string value to a document that did not have a valid value in that field. In this document, only a termination symbol was stored as a string value in DDB Cache (i.e., there is no space to store the new string). Thus, the new string is simply appended at a margin region of DDB Cache and the corresponding Auxiliary array is updated.

4.4. GPU Processing Using DDB Cache

We use a NVIDIA GPU for text processing with DDB Cache. We implement text processing GPU kernels by C language with CUDA (Compute Unified Device Architecture) platform [15]. For a regular expression matching, we implement a simple DFA-based GPU kernel similar to [8].

DDB Cache that implements a Value array (or a pair of Value and Auxiliary arrays) for each field is suited for GPU-based text processing. A string pattern matching that compares a given condition with a group of documents is performed as an independent thread. Thus, that for a large number of documents can be parallelized by utilizing a massive parallelism of recent GPUs.

A pattern matching result is returned from a GPU device to a host. The result is formed as a bitmap, in which each bit indicates whether each document satisfies a given condition or not. In CUDA, a memory is allocated in Bytes, and thus the matching results of eight documents are packed into a single Byte. If these eight documents that share the same result Byte are searched by multiple threads in parallel, their write operations to the result Byte will conflict and thus an atomic operation is additionally required. To avoid such a conflict, a single CUDA thread is created for each of eight documents that share the same result Byte. That is, a single thread performs a string matching that compares a given search condition with selected fields of eight documents stored in DDB Cache. Please note that further optimizations on the patten matching kernels are possible for some specific data types but such optimizations are beyond the scope of this paper and left as our future work.

The result transfer overhead from a GPU device to a host is quite small, because the result consumes only a single bit for each document. Actually, the result transfer time is shorter than that for a string pattern matching at a GPU device.

Because a text search for different documents can be performed independently, our DDB Cache approach is applicable for multiple GPUs cases by assigning different documents for each GPU. When n GPUs are available, DDB Cache is statically divided into n ranges, each of which is transferred to a corresponding GPU and searched independently. Each GPU processes the assigned range as well as the single GPU case.

5. Performance Evaluations

In this section, we evaluate the execution time of MongoDB queries using DDB Cache with a GPU or a CPU. It is compared with those of the original MongoDB without database indexes and that with database indexes. Please note that MongoDB is operated on a memory file system (i.e., tmpfs) for fair comparisons between MongoDB and our DDB Cache, because DDB Cache is running on a memory.

5.1. Evaluation Environment

Four types of queries are used for the experiments: 1) perfect-match of a single string field, 2) perfect-match of two string fields, 3) regular expression based text match of a string field, and 4) insertion of new documents. Documents are stored in MongoDB and searched by these queries. Although DDB Cache is applicable for the other data types (e.g., Integer and Timestamp), their evaluation results are similar to those of string queries; thus, their results are omitted in this paper.

MongoDB and our DDB Cache are operated at the same machine. The processor is Intel Xeon E5-2637v3 running at 3.5GHz and memory capacity is 128GB. A single NVIDIA GeForce GTX980 GPU is used with DDB Cache. Table 1 lists specification of the GPU. We use MongoDB version 2.6.6 and CUDA version 6.0.

Table 1. GPU spec. used in the experiments

	GeForce GTX 980
Number of cores	2,048
Core clock	1,126MHz
Memory clock	7,046MHz
Memory datapath width	256bit
Memory bandwidth	224GB/s
Memory capacity	4GB

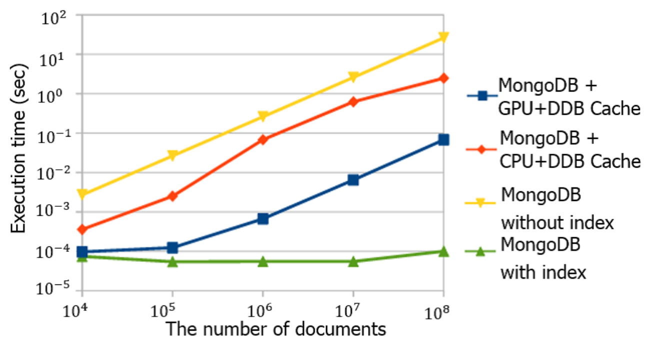

Figure 6. Execution time of perfect-match for a single string field

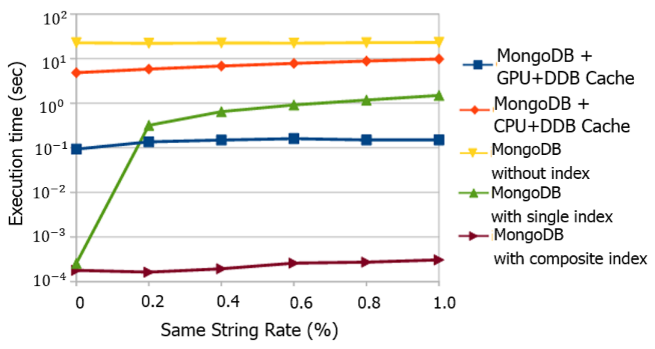

Figure 7. Execution time of perfect-match for multiple string fields

5.2. Perfect-Matching for Single String Field

MongoDB stores a number of documents, each of which has _id field and a single string field that contains randomly-generated eight characters. A perfect-match query for the single string field is performed for these documents. Below is an example of such a perfect-match query that retrieves documents whose “field1” field is matched to “abc.”

```
find({field1:"abc"})
```

Randomly-generated eight characters are used for the search condition.

Figure 6 shows the execution time of a perfect-match query vs. the number of documents in the database. Both X- and Y-axis are logarithmic scale. GPU+DDB Cache (DDB Cache with a GPU) always outperforms the original MongoDB without indexes. Its performance improvement is 458x when the number of documents is 100 millions. Next, we compare GPU+DDB Cache with the original MongoDB using indexes. When the number of documents is small, GPU+DDB Cache is comparable to MongoDB using indexes, while MongoDB using indexes outperforms GPU+DDB Cache as the number of documents increases. This is because the computational cost of indexed search is O(log n), while that with GPU+DDB Cache is O(n), where n is the number of documents.

5.3. Perfect-Matching for Multiple String Fields

Here, each document has _id field and two string fields that contain randomly-generated characters. A perfect-match query that specifies these two string fields is performed for these documents. Below is an example of such a perfect-match query that retrieves documents whose “field1” and “field2” fields are matched to “abc” and “xyz” respectively.

```
find({field1:"abc"},{field2:"xyz"})
```

Randomly-generated eight characters are used for these search conditions.

For the original MongoDB using indexes, we examine two cases: 1) a composite index that covers these two fields and 2) a single index only for the first field. We evaluate the single index case in addition to the composite index case, because a composite index that covers given fields may not always be available since the number of composite index combinations significantly increases as the number of fields increases, as discussed in Section 5.6.

For the single index case, the search execution time increases depending on the number of documents that have the same value in their first string field. In other words, the second fields of the documents that have the same value in the indexed field are linearly searched without indexes. Because the performance depends on the number of documents that have the same value in their first string field, here we define “Same String Rate” as a percentage of documents that have the same value in their first field. For example, when Same String Rate is 1% for 100 million documents, 100 unique values appear a million times, respectively. In this experiment, the number of documents is fixed to 100 millions while Same String Rate is varied.

Figure 7 shows the execution time of the perfect-match query for two fields vs. Same String Rate for 100 million documents. Y-axis (the execution time) is logarithmic scale. Except for the single index case, the execution time is almost constant regardless of Same String Rate; thus, there is no relationship between the execution time and Same String Rate in the other cases. In the case of MongoDB using the single index for the first field, its execution time is almost the same as that with a composite index when Same String Rate is 0%, while its execution time increases in proportion to Same String Rate. Our GPU+DDB Cache outperforms the single index case except when Same String Rate is quite low. Based on these results, we will discuss how to select GPU+DDB Cache and indexed search for incoming queries by considering Same String Rate in Section 5.6.

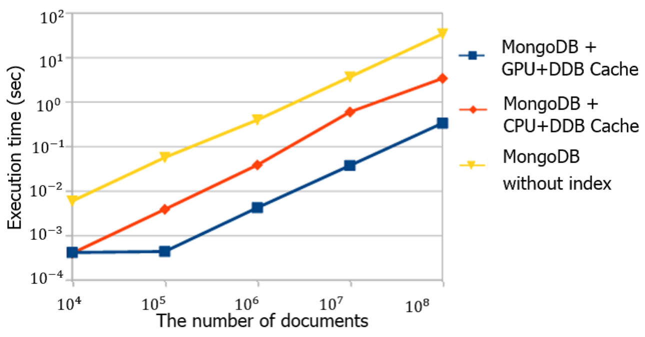

Figure 8. Execution time of regular expression match for a single string field

5.4. Regular Expression Matching for Single String Field

Here, each document has _id field and a single string field that contains randomly-generated 16 characters. A regular expression based match query that specifies a randomly-generated substring is performed for these documents. Below is a simple example of such a regular expression based match query that retrieves documents whose "field1" field contains a substring "abc."

```
find({field1:{$regex:/abc/}})
```

Randomly-generated eight characters are used for the substring.

As conventional database indexes cannot be used for such regular expression queries, MongoDB with indexes is not evaluated. We implemented a simple GPU kernel based on [8] for DDB Cache.

Figure 8 shows the execution time of the regular expression match query vs. the number of documents in the database. Both X- and Y-axis are logarithmic scale. GPU+DDB Cache outperforms MongoDB without index and it also outperforms CPU+DDB Cache (DDB Cache with a CPU) case. Its performance improvement is 101x and 10x compared to MongoDB without index and CPU+DDB Cache, respectively, when the number of documents is 100 millions. When the number of documents is 10 thousands, GPU+DDB Cache performance is almost the same as CPU+DDB Cache, because the number of documents is too small to utilize all the CUDA cores available in GTX 980 GPU.

5.5. Write Performance

MongoDB with indexes and our GPU+DDB approach are compared in terms of the write performance when inserting new documents. In addition to 100 million documents, each of which has six fields (_id field and five string fields containing eight characters), new documents are inserted to such documents in order to measure the write throughput.

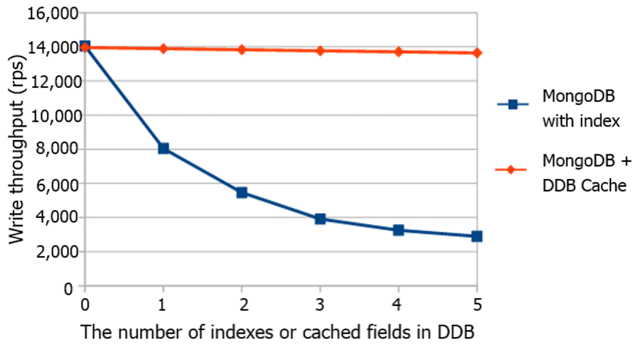

Figure 9. Write throughput (requests per second) vs. the number of indexes or cached fields in DDB Cache

Figure 9 shows the write throughput of MongoDB with indexes when the number of indexes is varied (_id field is indexed when X-axis equals zero). We use separated indexes for these fields rather than a composite index. It also shows the write throughput of GPU+DDB Cache when the number of fields cached in DDB Cache is varied (_id field is cached when X-axis equals zero). The results show that the write performance of MongoDB with indexes decreases as the number of indexes increases, while that of GPU+DDB Cache is almost constant regardless of the number of cached fields. Please note that if we consider the composite index, we need to take care of all the possible combinations of composite indexes, which may further degrade the write performance.

5.6. Comparisons to Database Indexes

Based on the above results, here we discuss how to select GPU+DDB Cache and indexed search in the document-oriented databases.

For regular expression based match queries or perfect-match queries in which indexes were not prepared, GPU+DDB Cache is the best solution. Obviously we focused on such cases in this paper. Here, we discuss the other cases as follows.

For perfect-match queries for a single field, using database index can reduce a search cost and outperforms GPU+DDB Cache, as shown in Section 5.2. However, we need to take into account a maintenance cost of the database indexes. Because entries in an index must be sorted, the B+tree based index is updated when a new document is inserted; thus the write performance is degraded, as shown in Section 5.5. On the other hand, because DDB Cache does not require any sorted data structures, its write performance is not degraded.

For perfect-match queries for multiple fields, GPU+DDB Cache, composite index, or single index should be selected carefully. Regarding the composite indexes, incoming query can utilize a composite index if an existing index covers all the fields specified in the query. The number of composite index combinations increases significantly as the number of fields increases, which may further degrade the write performance. Therefore, the composite index is the best solution when most queries specify the same set of fields. Otherwise, GPU+DDB Cache or single index should be selected.

As discussed in Section 5.3, our GPU+DDB Cache approach outperforms the single index except when Same String Rate is quite low. Again, the execution time of GPU+DDB Cache approach is constant regardless of Same String Rate, while that of the single index case increases as Same String Rate increases. More importantly, our GPU+DDB Cache approach does not affect a write performance, while an indexing degrades the performance.

5.7. CPU-GPU Data Transfer Time

Lastly, we discuss the CPU-CPU data transfer time. The CPU-GPU data transfer time to transfer 100 millions documents, which were used in Section 5.4, to the GPU is 0.21sec and it is corresponding to 64% of the execution time of a regular expression match query. However, please note that transferring the entire 100 millions documents to the GPU is required only when DDB Cache is created. When DDB Cache is updated, usually only the differences are transferred to the GPU; thus the CPU-GPU data transfer time will be quite smaller than 0.21sec. Even if we assume the entire 100 million documents are transferred to GPU at every time, GPU+DDB Cache approach is 62x faster than the original MongoDB.

If DDB Cache becomes too large and the GPU memory is not enough, the best way is to divide the DDB Cache into multiple portions (each of which can be fit into a single GPU memory) and store them in multiple GPU devices. If only a single GPU is available for use, DDB Cache is divided into multiple portions, and then we can repeat a procedure that transfers a portion to the GPU and performs a search for it. In this case, the CPU-GPU data transfer overhead is imposed every query.

6. Summary

Document-oriented databases are one of the most popular NOSQL databases due to their schema-less data structure and high horizontal scalability. MongoDB is the fourth most popular database in the DB-Engines Ranking [2] as of April 2015. Although a regular expression based text search is one of the attractive features of document-oriented databases, it incurs a significant computational cost because database indexes are not typically suited to such queries. This paper is the first paper that discusses how to utilize GPUs for document-oriented databases.

In this paper, we propose DDB Cache structure suitable for GPU-based text processing. By using GPU and DDB Cache, we can boost text search queries with modest modifications on the original document-oriented database.

Our GPU and DDB Cache approach is useful especially for text search queries that cannot utilize the database indexes. Experimental results using GeForce GTX 980 show that our approach improves the performance of regular expression search queries by up to 101x compared to the original document-oriented database.

Acknowledgements This work was supported by SECOM Science and Technology Foundation and JST PRESTO.

References

[1] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, August 2013.

[2] “DB-Engines Ranking,” http://db-engines.com/en/.

[3] “MongoDB,” http://www.mongodb.org.

[4] “Apache Couch DB,” http://couchdb.apache.org.

[5] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava, “Using q-grams in a DBMS for Approximate String Processing,” *IEEE Data Engineering Bulletin*, vol. 24, no. 4, pp. 28–34, Dec. 2001.

[6] A. Korotkov, “Index Support for Regular Expression Search,” in *The PostgreSQL Conference (PGCon’12)*, May 2012.

[7] J. Cho and S. Rajagopalan, “A Fast Regular Expression Indexing Engine,” in *Proceedings of the International Conference on Data Engineering (ICDE’02)*, Mar. 2002, pp. 419–430.

[8] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, “Parallelization and Characterization of Pattern Matching using GPUs,” in *Proceedings of the International Symposium on Workload Characterization (IISWC’11)*, Nov. 2011, pp. 216–225.

[9] J. Naghmouchi, D. P. Scarpazza, and M. BereKovic, “Small-ruleset Regular Expression Matching on GPGPUs: Quantitative Performance Analysis and Optimization,” in *Proceedings of the International Conference on Supercomputing (ICS’10)*, Jun. 2010, pp. 337–348.

[10] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong, “GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’12)*, Feb. 2012, pp. 129–140.

[11] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O’Connor, and T. M. Aamodt, “Characterizing and Evaluating a Key-value Store Application on Heterogegenenous CPU-GPU Systems,” in *Proceedings of the International Symposium on Performance Analysis of System and Software (ISPASS’12)*, Apr. 2012, pp. 88–98.

[12] S. Morishima and H. Matsutani, “Performance Evaluations of Graph Database using CUDA and OpenMP-Compatible Libraries,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 4, pp. 75–80, Sep. 2014.

[13] P. Bakkum and K. Skadron, “Accelerating SQL Database Operations on a GPU with CUDA,” in *Proceedings of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU’10)*, Mar. 2010, pp. 94–103.

[14] “The MongoDB 3.0 Manual,” http://docs.mongodb.org/manual.

[15] “NVIDIA CUDA,” https://developer.nvidia.com/cuda-zone.