Making Cheetah Faster

Alan Ng, Gary Chaw, Dustin Kut Moy Cheung

Department of Electrical and Computer Engineering
University of Toronto
{alan.ng, gary.chaw, dustin.kutmoycheung}@mail.utoronto.ca

Abstract—In this report, we describe how we modified Cheetah, an in-memory database, to speed up queries and insertions using threads. The queries and inserts are transformed into tasks in a queue and the threads in a threadpool pull tasks to execute. Our results show a speedup in query time as the number of threads increase when using the column store. However inserts suffer from the global lock imposed on the store which results in poor performance. Our results show that it is possible to improve the performance of Cheetah by adding more threads and exploiting the different cores in a multi-core processor. However we are unable to explore the impact of cache locality in our design to due to limited tools in the Java ecosystem.

1 Introduction

The need for in-memory databases has exploded in recent years with the requirement for faster data access to provide near real-time data to users. These types of databases are challenging the dominance of traditional relational databases in an era where multi-core processors and main memory are becoming cheaper.

Unlike traditional databases, in-memory databases do not require optimization of hardware for disk storage, or tweaks to the underlying operating system, allowing an extremely high throughput rate for writes and access using default configurations. In-memory databases do not suffer from unpredictable performance experienced by traditional databases as the seek time required to read data from a hard-drive is eliminated.

Typically in-memory databases are employed when huge amounts of data need to be queried and processed in real-time. An example is a real-time feed of tweets having a specific hashtag for users of the Twitter platform [1] which requires scanning through millions of tweets posted on Twitter.

Together with the explosion of in-memory databases is the emergence of the JSON data format. JSON has replaced XML as the format of choice for serializing data to be sent in between servers because of native JSON parsing abilities in the Javascript language [2]. Unlike relational databases, some NoSQL databases allows JSON as input into their system. Examples include MongoDB, CouchDB and Elastic Search. The exploration of mapping JSON data into memory and running queries in parallel still remains an interesting new topic to research to find out if we can optimize the architecture to achieve higher throughput.

2 Background

Cheetah inserts data into its store by reading JSON data saved in a file. The JSON data consists of arrays of objects, where each object consists of many name/value pairs. Each of these names might contain values of type: boolean, arrays, number, strings, and other objects. The names in each object are often different from each other, making them unsuitable to be stored in a relational database because of the sparseness of the data.

In a relational database, data is organized into tables, where each table consists of columns. Each column has a particular data type associated with it. The global configuration of tables and the column types is known as the schema of the database.

If one would like to store all of the objects in the JSON data in a traditional database, one of the strategies would be to save the objects into one table, where each column represents all the potential names that the objects can have, and each row represents the object itself. However, this causes huge gaps in our table since most of the time only a few names are present in an object, causing inefficient use of vital memory space.

The Cheetah design explores alternate mappings of JSON data into main memory while making use of relational database paradigms to save data. We describe below the three mappings used in Cheetah to insert the JSON data into the database.

2.1 Column Store

The column store creates a new table for each name in the object if the table has not been created yet. The table name is the JSON key whose value needs to be saved. To know which value is linked to which object, every object being saved is assigned a unique object id, and each value is then mapped to an object id. As such, the table has two columns, one for the object id and one for the value. That means that the data of an object is broken down and saved in different tables, with the object id as the only connection back to the object.

Those columns are implemented internally by using primitive arrays. The main disadvantage of primitive arrays is that we have to determine beforehand how big the arrays should be, which is not very practical in real-world applications. Other data structures are explored later in this report.

The column store performs better memory-wise compared to the other stores if the objects being stored have sparse names/keys. However, searching data requires more time because we have to scan through all the tables in the store to see if each table has a value for a particular object id.

It would also be interesting to study the impact of spreading the JSON data into different tables on cache locality. In theory, this scheme would affect the cache locality because of the fragmentation of data for a particular object in different memory locations. However, we cannot explore deeper on cache locality due to the lack of appropriate tools to measure cache hit/miss in the JVM platform.

2.2 Row Store

As opposed to the column stores strategy of using multiple tables, only one table is used in the row store. Each row in the table is mapped to a schema which indicates which row points to which name in the object. For example, if there are five names listed in the schema, the first object will store its values in row 1 to row 5, and the second object will store its values in row 6 to row 10 etc.

This scheme is great for searching, since we just need to scan a table to find a particular object with all its name/value pairs. In theory, this greatly improves cache locality. However, this is not very memory efficient since the JSON objects that we save are typically sparse, which means that a lot of rows in our table will be empty, due to a lack of a value for a particular name.

2.3 RowCol Store

The RowCol store is a hybrid store that combines both the row and column store philosophies. It consists of tables, where each table manages a subset of names indicated by the schema for that particular table. This design shares the pros and cons of both the row and column store. It tries to reduce the sparseness of data, while keeping the data not as fragmented into different tables as in the column store. However, there is a need to know which JSON names will potentially be stored beforehand due to the schema requirement for each table.

2.4 Current Flow of Simple Query Executor

The Simple Query Executor program uses Cheetahs store engine to store sparse data and run queries on it. The program reads in the JSON data and stores the data depending on the chosen store type: row store, column store, or row-column hybrid store. Then a batch of SQL read-queries are executed. Each query is read and performed sequentially. For each query, the Simple Query Executor program parses the query, then calls the store engine to execute the query. When the store engine executes the query, it overwrites a ResultSet object. The program then prints the results and brief runtime statistics based on what is stored in the ResultSet. This process is repeated for all subsequent queries, which clear and overwrite the ResultSet. Figure 1 below illustrates the program flow:

2.5 Threads

A thread of is the smallest executing context provided by an operating system. At its lowest level, a thread maintains an independent set of values for the processor registers, such as the program counter. That means that threads executing in the same process can potentially share resources such as memory, and instruction code, while executing different parts of the code due to each thread having their own program counter. A single threaded processor switches between different threads by time-division multiplexing to give the illusion of threads running at the same time for the user. More

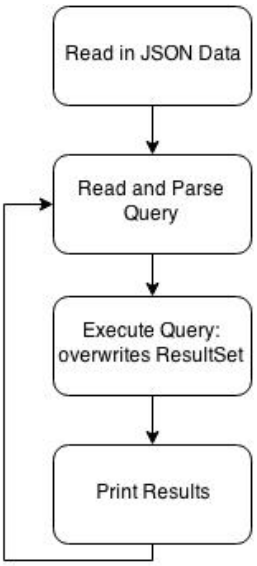

Fig. 1. Flow of sequential Simple Query Executor

recently with the popularity of multiprocessors, each thread can truly run at the same time by being executed by different processors at the same time.

To run a thread in Java, we can either declare a class that extends the Thread class, or have the class implement the Runnable interface. The advantage of using the Runnable interface is that a class can implement multiple interfaces, but can only extend one class. As such, using the Runnable interface has the advantage of requiring less modifications in their existing class. Another advantage of the Runnable interface is that it can be used directly with Javas implementation of a Thread pool.

2.6 Queries

Cheetah supports SQL statements for querying data. The querying engine allows users to select some keys/names out of objects, and optionally provide restrictions where a particular key/name must be of a certain value or range of values while searching for objects. The engine can also return a count of objects that match a certain restriction, Some query types are not supported in Cheetah, like the NOT operator, and JOINS where different objects can be combined together if one or more of their names/keys match. Currently the UPDATE statement is also not supported.

When queries are done without INSERT or UPDATE statements in between, the use of threads while querying could potentially offer huge speedups since no locks are needed while reading from the arrays. However, an INSERT statement will impose a lock on the underlying data structures to avoid any race conditions, and provide the most up-to-date data for the next query. This could potentially affect the performance of threads in that situation.

3 Analysis of the Store Structure

As mentioned before in the column store subsection, a table will contain two columns, one for the object id and one for the value, both implemented using primitive arrays.

There is an intrinsic connection between the object id and the value, where the object id is the key to the value. As such, it is natural to explore other data structures that implement a key-value structure, like a Map or a Hash.

We therefore explore the use of Javas HashMap and TreeMap (which is implemented as a red-black tree, which is always balanced) as alternatives to the array structure. While performing some benchmark tests on these alternate data structures, we notice a huge performance drop.

[show the results below]

We can see that both the HashMap and the TreeMap take a significant amount of time to insert and read data. The HashMap insertion and reading are expensive since a hash value has to be computed every time.

As for the TreeMap, the insertion is significantly more expensive than reading because each insertion requires a new Tree node object to be created, which is expensive.

Even though in theory, using a Map or a Hash has a architectural desirables, in practice, the overhead required to maintain these structures is very expensive compared to using a simple primitive array. As such, we decided to keep using arrays in our stores for performance reasons, even though the array structure has the disadvantage of being fixed in size.

4 IMPLEMENTATION

For the remainder of this report, we will only focus on working with the column store and the new parallelized version of the column store. We focused on the column store because of being excellent memory-wise, while being flexible enough to not require any schema to manage the table layout.

4.1 Parallelization of Read Queries

There are several modifications to the Simple Query Executor and to the store engine in order to support parallel queries. The changes only pertain to query execution. A thread-based approach is chosen to parallelize read-queries. We use the Thread pool implementation provided by the Java standard libraries [citation needed] to re-use existing threads to parallelize queries. Each query is transformed into a task that is then pulled by a free thread in the thread pool. The program modifications have to handle several issues in parallelizing Cheetah. Two different program flows were considered to tackle parallelization of read queries. Of the two variants, only one is evaluated.

4.1.1 Issues with Parallelization

In the sequential program, each query is performed one after the other. The ResultSet object is cleared and rewritten with each query. For parallel read queries, this ResultSet will need to be multiplied by the number of threads in the threadpool. These result sets were also very large in size. Each result set contained 3 arrays (long, int, and String arrays) and each array pre-allocated space for 100M elements. Given these large arrays, it is very easy to run out of memory on the machines we ran these tests. We could allocate more swap but this would defeat the purpose of developing a program for in-memory databases.

4.2 Parallel Program Flow

In the parallel program flow, the Simple Query Executor still starts by reading in the JSON sparse database. Using the thread-based approach, the idea is to have each Query object executed as a runnable object. The program initializes a fixed thread pool that will handle query execution. From this point, two variants of the program are explored to parallelize the Query execution.

In the first variant, an additional thread is created. This thread holds the print queue which accepts queries that have finished execution and will print out results based on the result sets contents. The query execution itself handles parsing of the query string and execution of the query. This method was deemed not feasible since it requires copying and storing the result set into the query and passing it to the print queue. At the speed the queries were executing, the program easily ran out of memory. Figure 3 below outlines the program flow for the first variant.

In the second variant, for each thread in the threadpool, an associated ResultSet object is initialized in the store engine. The store engine is modified to contain a map of thread id and result set associations. This initialization was chosen to avoid having each query instantiate their own result sets. Since result sets each contain 3 large arrays, doing so would have been costly. There is no print queue thread in the 2nd variant. Instead, the query handles parsing of the query string, execution of the query, and computing the string that needs to be printed based on the result set. By adding this string computation, we decrease the memory requirements for the query from a large result set to a small string. We also increase the time it takes to finish a query. However, this time was also additional in the original sequential program. Once all the queries have terminated execution, the results from each query are printed one after the other. The figure XYZ below illustrates the program flow with the result string computation.

The second method above is chosen for our experiments.

4.3 Parallelization of Writes

4.3.1 Enabling Inserts

The Cheetah implementation provided to us did not support transactions other than read query, select. In order to explore and experiment in-memory database with read/write mixed workloads as well

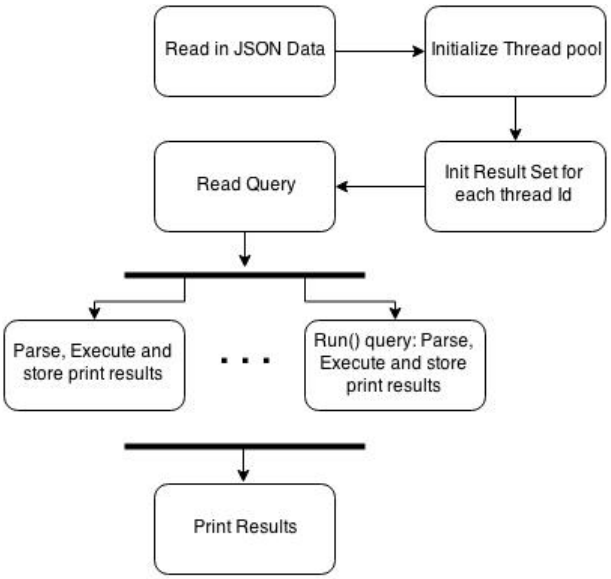

Fig. 2. Parallelized Query Execution with Print Queue

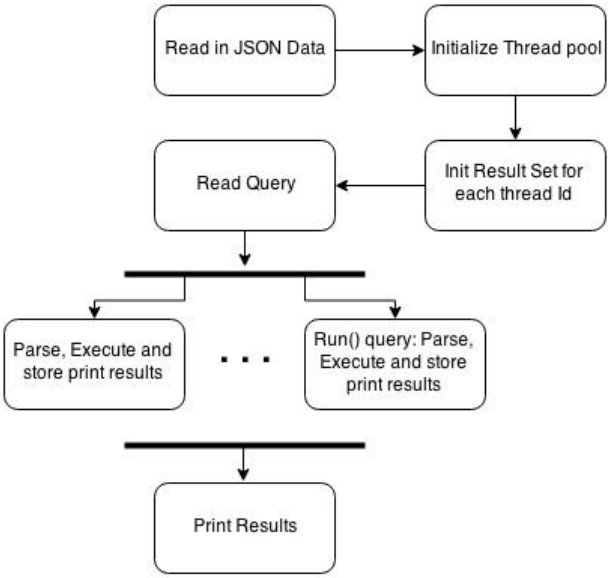

Fig. 3. Parallel query execution with result string computation for smaller memory footprint

as concurrency, we have decided to add support to the INSERT SQL statement, a write transaction.

The insert statement adds a new record into the data store from a JSON object, which may be a tree-structure object, comparing to traditional SQL-like insert, which specifies the new record in flatten structure of key-value pairs.

[add an example of insert statement with sql object]

Similar to read queries, we have enhanced Cheetah to support parallel execution of write transactions.

4.3.2 Locking

Concurrency problems such as lost update, uncommitted dependency and inconsistent analysis can be solved by implementation of locking. We have explored the following two locking models and chose to implement the first as our locking protocol.

Simple lock: A mutex lock is placed to protect the data so that a write transaction needs to acquire lock on the data before it writes the data. Therefore, any addition and modification of elements into the data store must be inside a lock-protected critical region to prevent concurrent writes to the same data. However, read transaction is not required to

acquire the lock on the data.

Reader-writer lock: readers block writers but permit other readers, and writer blocks readers and writers. A read transaction will acquire shared Lock on the data before it starts reading data. And a write transaction will acquire exclusive lock before it starts writing data.

Different from update statement which modifies one or more existing entries in the data store, insertion statement simply appends new entries to Cheetah data store. Since there is no other write operation other than insertion is allowed in current Cheetah implementation, we aim to protect the potential conflict by concurrent insertions by two or more threads.

4.3.3 Lock Granularity

Coarse-grained lock: A single lock at data store (table) level. Any insertion of a new record will result in locking the entire data store. If another insertion needs to add a different record to the data store, it is forced to wait until the first transaction exits the critical region.

In Java, A ReentrantLock is created when the data store is initialized. Lock is acquired by the thread before the JSON object is inserted to the data store, inside InsertObject() function call. And the lock is released after the entire JSON object has been inserted.

Fine-grained lock: One lock per data column in data store. Instead of having just one single lock protecting the entire data store, there will be one lock per data column

In Java, A ReentrantLock is created when the column object is initialized. Lock is acquired by the thread before the column value of the inserted record is added to the column store, in saveStrValue(), saveBoolValue(), saveDoubleValue(), saveLongValue().

4.3.4 Support of other types of write transactions

Due to limited time span of this project, we did not implement functionalities to support other types of write transactions such as update, which modify one or more existing records, and delete, removal of existing records in data store.

5 Experimental Results

5.0.5 Parallel Read Queries

5.0.6 Parallel Inserts

5.1 Lock Granularity

6 Related Work

Its always interesting to investigate the approaches of other in-memory databases used in the wild. Redis is one of them. Redis is an open-source cache and store that is used extensively in the industry. Even though Cheetah stores JSON data, the JSON format is still very similar to a key-value format, where each attribute in the JSON object is a key. Redis is famously single-threaded and deal with all network query (and write) requests by using non-blocking IO coupled with an event-loop approach, which allows Redis to have a high throughput. There is a possibility to run Redis in a multi-process fashion by using Redis Cluster [citation needed]. This allows the Redis database to be spanned across several machines.

Redis uses an adaptive scheme to save their key-value data. The most natural way of storing this data would be by using a highly-efficient hash structure.. However, for small key-value storage, they instead instead just encode them in an O(N) data structure, like a linear array with length-prefixed key value pairs [http://redis.io/topics/memory-optimization]. the hash will be converted into a real hash table as soon as the number of elements it contains will grow too much. since a linear array of key value pairs happens to play very well with the CPU cache (it has a better cache locality than a hash table). [same citation as above].

On the data partitioning side, HYRISE is another in-memory database system that adopts a hybrid system for storing data. It dynamically partitions tables of varying widths depending on the most accessed columns. This contrasts with Cheetah which requires a schema to determine the layout of its tables for both the row and the rowcol store. HYRISE will alter their partitioning based on the nature of the queries. If there are queries that perform sequential scans (for example, to find values of a column who are within a range), they determined that narrow partitions perform better because of the improved cache locality. For operations that require a lot of inserts, updates or deletes, a wider partitioning is favored. Using a model that is able to predict the performance of the different partitionings,

HYRISE will change the partitioning according to the nature of the queries. [[cite the hyrise paper]

The Argo Document store [citation needed] is the closest architecturally to the functionality and store design of Cheetah. They employ a very simple store that deconstructs the attributes in JSON data into keys and use a single table to store the object id, the key and its value. Similar to Cheetah, it also supports querying via SQL statements and supports INSERT, DELETE, JOINS, and SELECTs. Argo also uses threads while querying with JOINs, but doesnt use threads for querying in general.

7 Conclusion and Future Work

Use of threads has significantly improved the time it takes to run queries. We are able to improve the performance while keeping most of the Cheetah architecture relatively unchanged. [talk about writes] We also explored the various data structures that we could use in our table before using primitive arrays due to its superior performance for inserts and reads.

Future work needs to be done to experiment with different JSON data mappings to memory to exploit multi-threading while preserving cache locality. Due to lack of existing analysis tools for the JVM platform, this task remains hard to do. Finally, the two stores not modified in our work, namely the row store, and the rowcol store, could be modified using the same architectural changes we performed for the column store, to exploit multi-threading and observe how these stores perform in these situations.

The use of different types of queries, and their impact on the multi-threaded design should also be explored in the future. In our current work, we only performed queries on SELECT WHERE types. Other query types could provide better insight on the behaviour of threading in the current data stores, and perhaps provide hints on a potential new store configuration that could provide better gains for threaded queries.

Acknowledgments

The authors would like to thank...

References

[1] "How twitter uses redis to scale - 105tb ram, 39mm qps, 10,000+ instances." http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html.

[2] "Json: The fat-free alternative to xml." http://www.json.org/fatfree.html.