A Demand-Driven Bulk Loading Scheme for Large-Scale Social Graphs

Weiping Qu and Stefan Dessloch

University of Kaiserslautern
Heterogeneous Information Systems Group
Kaiserslautern, Germany
{qu,dessloch}@informatik.uni-kl.de

Abstract. Migrating large-scale data sets (e.g. social graphs) from cluster to cluster and meanwhile providing high system uptime is a challenge task. It requires fast bulk import speed. We address this problem by introducing our "Demand-driven Bulk Loading" scheme based on the data/query distributions tracked from Facebook's social graphs. A client-side coordinator and a hybrid store which consists of both MySQL and HBase engines work together to deliver fast availability to small, "hot" data in MySQL and incremental availability to massive, "cold" data in HBase on demand. The experimental results show that our approach enables the fastest system's starting time while guaranteeing high query throughputs.

Keywords: Bulk loading, HBase, MySQL.

1 Introduction

As the biggest social network company, Facebook's social graph system nowadays serves tens of billions of nodes and trillions of links at scale [1]. Billions of daily queries demand low-latency response times. Recently, a social graph benchmark called LinkBench [2] was presented by Facebook which traces distributions on both data and queries on Facebook's social graph stores.

Two main tables `node(`<u>`id`</u>`, type, data)` and `link(`<u>`id1, link_type, id2`</u>`, data)` are used to build the social graph at Facebook (primary keys are underlined). Nodes represent objects like user, photo, video, etc. while links are connections between the objects and have types like "post", "like" and "friend_of". We learned several interesting facts from LinkBench, for example, one observation on access patterns and distributions states that there is always some "hot" data that is frequently accessed while massive amounts of "cold" data is seldom used. With a 6-day trace, 91.3% of the data is cold. In addition, hot data often exists around social graph nodes with high outdegrees, which means the access likelihood grows along with the node outdegrees. As an example, a video with high *like* rates will be recommended more widely than others. Based on another observation on social graph operations, an operation called *get_link_list* occurs frequently and constitutes 50.7% of the overall workload. The signature of this operation is `get_link_list(link_type, id1, max_time, limit, ...)` where id1 is the starting node id of this link. Given the type of a link (e.g. *like*) and the id of the starting node (e.g. a user *id1*), *get_link_list* returns a list of links to answer certain types of queries like "what objects have been recently *liked* by user *id1*". This *get_link_list* performs a short range scan in massive amounts of rows in graph stores based on (`link_type` and `id1`), a subset of the composite key.

As more and more applications are added to Facebook, like Facebook Messaging and Facebook Insights [3], and workloads evolve on graph stores, the change of the underlying data infrastructure or software requires migrating existing data from cluster to cluster while high data availability must be guaranteed. To provide 24-h system uptime, normally hot-standby clusters are used to store replicas of source data and serve query workloads during data migration. This incurs high data redundancy and the new system still cannot start until a long-running bulk import is finished.

Our work investigates the problem of migrating large-scale social graphs based on their data distributions and access patterns introduced above. To guarantee high system uptime, a trade-off between data availability and query latency is utilized in this work. The "hotness and coldness" of migrated data is balanced by a hybrid graph store which is composed of a traditional index-based relational MySQL (http://www.mysql.com) database and an Apache HBase (http://hbase.apache.org) cluster. Both systems have received high attention as a backend storage system for realtime operations on big data. The debate on MySQL and HBase began in 2010 in terms of multiple metrics like read/write throughput, I/O, etc. In this work, we will first compare these two systems regarding their bulk load speed and short range scan latency and then introduce our "demand-driven bulk loading" scheme.

The comparison of MySQL and HBases' load and scan performance is given in Section 2 and the motivation of this work is explained there. In Section 3, we introduce the architecture of our "demand-driven bulk loading" scheme. The experimental results are analyzed in Section 4. We discuss related work in Section 5, and Section 6 concludes our work.

2 Bulk Loading in MySQL and HBase

In [4,5], the performance of sequential/random read/write access has been compared among Cassandra, HBase and (sharded) MySQL. The results show that, due to their different architectures, HBase has the highest write throughput when the insertions fit in memory, while MySQL is best for read access. Both engines provide extra bulk load utilities in addition to the interfaces for individual, row-level writes/updates. In this section, we compare their bulk load mechanisms by analyzing their architectural differences. Based on the comparison result, we describe our motivation of providing incremental availability to external queries during bulk loading large-scale social graphs.

2.1 MySQL

Like other traditional databases, MySQL uses B-trees as an index structure for fast read and write on large tables. One crucial step of generic bulk loading in traditional databases is an index construction process. Using a classical sort-based bulk loading approach, the entire data set is pre-sorted (O(nlog(n))) and grouped in file blocks as index leaf nodes. A B-tree index can be easily built from this set of sorted leaf nodes in a bottom-up fashion from scratch. In contrast, inserting the tuples from the same data sets once at a time in a top-down fashion without pre-sorting incurs overhead i.e. a lot of splits on index's internal nodes and a large number of disk seeks with random I/O. There are other approaches for building indices during bulk loading, like buffer-based/sample-based bulk loading [6] which will not be detailed here.

To import large amounts of data in MySQL, there are two primitive approaches: batch insert and bulk loading. By including multiple tuples in one `INSERT` statement and inserting these tuples in batch, batch insert results in fewer operations and less locking/commit overhead. Yet the bulk load command `LOAD DATA INFILE` is usually 20 times faster than using the `INSERT` statement because of its less overhead for parsing [7]. However, the user has to ensure that the tuples to be inserted won't violate integrity constraints. Before bulk loading, the use of indices is normally disabled to avoid disk seeks for updating on-disk index blocks at load time. After bulk loading, indices are enabled again and created in memory before writing them to disk. However, when bulk-loading non-empty tables where indices are already in use, a performance impact of bulk-loading on concurrent reads occurs.

As mentioned in Section 1, the frequently used *get_link_list* operation performs a short range scan on a subset of the composite key. Therefore, using MySQL as the graph storage backend, very low scan latency can be achieved by traversing the leaf nodes of primary key index sequentially after the starting block has been found.

2.2 HBase

Apache HBase^{TM} is the Hadoop database built directly on the Hadoop Distributed File System (HDFS) [8]. HDFS is an open-source version of Google File Systems (GFS), which inherently provides batch processing on large distributed files using MapReduce jobs. HBase was modeled after Google's Bigtable [9] and provides random, realtime read/write access to HDFS. This is done by directing client requests to specific region, with each server handling only a fraction of large files within a certain key range.

Both Bigtable and HBase use an "append-only" log-structured merge tree (LSM-tree) [10] structure. In HBase, inserted tuples are first buffered in an in-memory structure called MemStore and sorted very fast in memory. Once the size of the MemStore exceeds a certain threshold, it is transformed to an immutable structure called HFile and flushed onto disk. A new MemStore is then created to further buffer new incoming rows. Updates on existing rows are treated as new insertions appended to existing files instead of in-place modification, which needs random disk seeks for reading updated blocks. In addition, no auxiliary index structure needs to be maintained. In this way, high write throughput can be achieved in HBase. However, a single row can appear many times in multiple HFiles or MemStore. To retrieve a single row, HBase has to scan those HFiles or MemStore that contain the copies of this row and merge them to return the final results. Sequential scans are carried out on sorted HFiles. Thus the read speed is dominated by the number of files in a region server. In order to improve read performance, a compaction process runs periodically to merge HFiles and reduce the number of row copies. Furthermore, Bloom filters can be used to skip a large number of HFiles during reading.

To insert large amounts of files into HBase, an efficient MapReduce-based bulk loading approach can be used to directly transform HDFS files into HFiles without going through the write path introduced above. Each row appears only once in all HFiles. The map tasks will transform each text line to a Put object (a HBase-specific insert object) and send it to a specific region server. The reduce tasks sort these Put objects and generate final HFiles. This process is much faster than HBase writes as it exploits batch processing on HDFS. The *get_link_list* operation can benefit from this bulk loading approach as well since the number of HFiles to read is small. By setting up Bloom filters, target HFiles can be found very fast.

2.3 Motivation

As introduced in Section 1, 91.3% of the social graph is rarely used while 8.7% of the data sets are frequently accessed. When migrating such a social graph to new clusters, the availability of hot data is delayed since the system downtime will only end when all data has been loaded. For frequently emerging queries, system uptime could start earlier if there was a mechanism that can tell whether all the relevant data is already available before the remaining data is loaded. As data is loaded chunk by chunk, it would be enough to have a global view of the key ranges of all the chunks before starting loading. This can be seen as an index construction process. In this way, a query can identify its relevant chunks by comparing its queried key with the key ranges of all the chunks. Once its relevant chunks have been loaded, this query can start to run over the current storage state. In addition, as the small amount of hot data can be arbitrarily distributed among all the chunks, faster availability to frequent queries can be achieved by prioritizing the loading of the chunks which contain hot data.

MySQL's bulk loading builds indices either during data loading or after loading. Building indices upfront is impossible. Using HBase's bulk loading, source files must be first copied to HDFS and then transformed to HFiles. Comparing the bulk loading techniques in both systems, similar performance can be expected, since both techniques have to sort files either to build B-tree index in MySQL or to generate HFiles in HBase. But HDFS's batch processing feature can be exploited in HBase's two-phase bulk loading approach to build indices on copied chunks in batch upfront before going to the second transformation phase, which is more desirable.

However, according to the experimental results in [2], MySQL slightly outperforms HBase in latency and MySQL executes *get_link_list* operations 2x faster than HBase. We see a trade-off between fast availability and low query latency here. Loading all data in HBase can have fast availability by creating indices upfront. Loading all data in MySQL leads to low query latency after long-time bulk loading ends. It makes sense to load only a small amount of hot data into MySQL in a smaller time window for fast processing while copying massive amounts of cold data into HBase where its query latency is still acceptable for cold data. But the cost of identifying the hotness and coldness of tuples could be a large overhead for bulk loading. Hence, we introduce our "demand-driven bulk loading" scheme to address these considerations.

3 Demand-Driven Bulk Loading

In this section, we introduce the architecture of our demand-driven bulk loading scheme. According to the modification timestamps of files, the whole input data set is separated into two parts: small number of recent files and large, historical files. Recently changed files are imported into a MySQL table called "link table" using MySQL's own fast bulk load utility. These files are used for answering queries on hot data and providing partial results for all other queries. Meanwhile, massive amounts of historical files are first split to chunks with pre-defined size and then copied into HDFS in batch. With parallel loading of recent and historical files into MySQL and HDFS, respectively, the latency of loading HDFS is normally higher than that of MySQL's bulk load due to the input size, thus dominating the overall load speed. After loading in MySQL completes, hot data is available for querying. The HDFS files will be gradually loaded into HBase to complement the MySQL query results for answering queries that involve cold/historical data. Figure 1 illustrates the architecture of our hybrid-storage approach.

Two main processes are involved in this hybrid-storage architecture: offline index building and online bulk load coordination. In contrast to traditional bulk load approaches, client requests are allowed to query link data before the historical data sets are completely available in HBase. To determine the completeness of query results on the client side, a so-called *bucket index table* is used. At the HBase layer (left) side of this architecture, an offline MapReduce job called *distribute chunk* (*dist_ch*) job is batch processed on each file chunk in HDFS once copied from the remote server by a *HDFS loader*. The implementation of this job is based on a hash function that maps and writes each text line in a chunk to a specific "bucket" (HDFS file) with a unique id and a key range disjoint from others. A new bucket will be created if needed and its *bucket index* and *key range* will be captured by the bucket index table at the (right-side) MySQL layer. These steps form the *offline index building* process. More details will be provided in Subsection 3.1. With the completion of the last *dist_ch* job, all cold

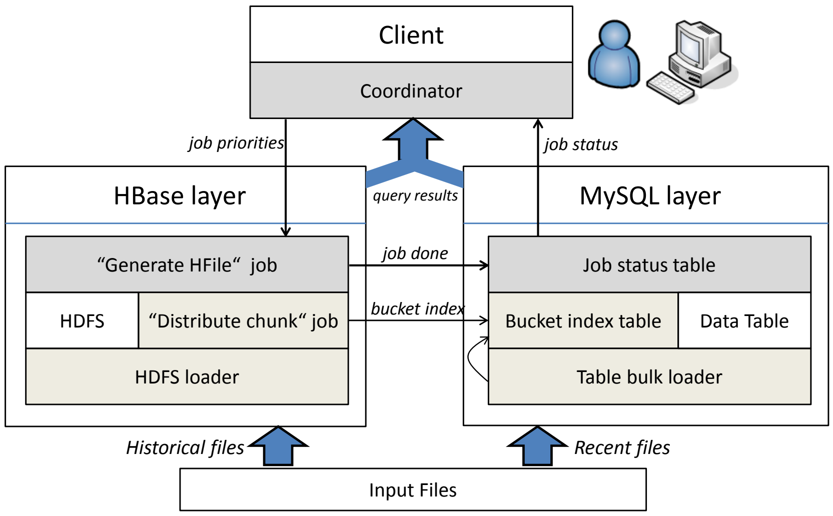

Fig. 1. Architecture of Demand-driven Bulk Loading

files have been copied into HDFS and clustered into multiple buckets with disjoint key ranges. The key ranges and index information of all the buckets are contained in the bucket index table in MySQL.

At this time, system uptime begins and query requests are allowed to run on our hybrid link data storages through a client-side *coordinator* component. At the same time, another online MapReduce job called *generate HFiles* (*gen_HF*) job is triggered to continuously transform buckets in HDFS to HFiles so that they can be randomly read from HBase. The transformed HFiles incrementally build a "link table" in HBase which can be seen as an external "table partition" of the "link table" in MySQL. Tuples stored in both engines share the same logical table schema. With a given key (the id1 of the starting node of a link) specified in a query, the coordinator checks whether the HBase layer should take part in this query execution by asking the MySQL-side bucket index table. If so, another *job status table* tracks the availability of the required tuples in HBase. For tuples available in HBase, the query request is offloaded to HBase by the coordinator. In case there are tuples that are not available yet because they reside in buckets that wait in the *gen_HF* queue, the query is marked as "incomplete" and buffered by the coordinator. As more and more incomplete queries occur at the coordinator side, the coordinator makes the online MapReduce job prioritize the job execution sequence for specific buckets, delivering fast availability on demand. Once the buckets are transformed to the portions of HBase's "link table", corresponding buffered queries are released. This process is called *online bulk load coordination*. The implementation of this process will be detailed in Subsection 3.2.

3.1 Offline HDFS Load and Index Construction

We use a dedicated Hadoop cluster to take over the job of loading massive amounts of cold/historical link data from remote servers to MySQL. HDFS's free copy/load speed and batch processing (using MapReduce) natures are exploited here to provide only indices on clustered file groups (buckets) using the *dist_ch* jobs, as introduced above. A *dist_ch* job writes text lines in each chunk in HDFS to different buckets (HDFS directories) and outputs indices of new buckets to MySQL's bucket index table (Hadoop's MultipleOutputs is used here to include TextOutputFormat and DBOutputFormat for writing lines to buckets and writing indices to MySQL, respectively).

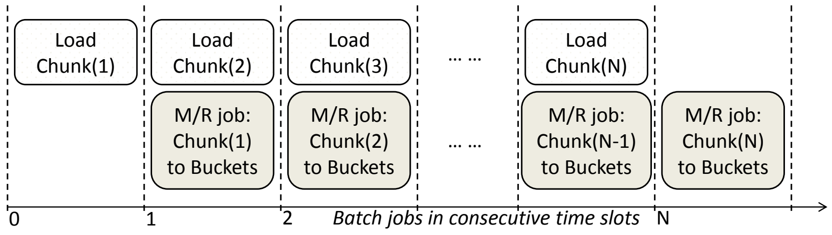
Load Chunk(1)
Load Chunk(2)
Load Chunk(3)
... ...
Load Chunk(N)
M/R job: Chunk(1) to Buckets
M/R job: Chunk(2) to Buckets
... ...
M/R job: Chunk(N-1) to Buckets
M/R job: Chunk(N) to Buckets
0
1
2
Batch jobs in consecutive time slots
N

Fig. 2. Execution Pipelines in the HDFS loader

Instead of running one big MapReduce-based *dist_ch* job after all files have been completely copied from a remote server, large historical files are split to several chunks and multiple small *dist_ch* jobs are executed in parallel with copying small file chunks to HDFS. As shown in Figure 2, chunk copying and *dist_ch* job run simultaneously in each time slot except the first and the last one which copies the first chunk and builds the indices for the last chunk, respectively. The resource contention is low since chunk copying does not use any MapReduce job and each *dist_ch* job runs individually. As the chunk copying pipeline overlaps the *dist_ch* job pipeline, the overall latency is derived from loading all chunks plus running the last *dist_ch* job. The chunk size is selected in a way that the latency of loading a chunk of this size is higher than running one *dist_ch* job on that chunk. If this requirement can be guaranteed, the chunk size should be defined as small as possible so that the time running the last *dist_ch* job is the shortest. Therefore, the overall system downtime is similar to the latency of copying massive amounts of cold data from a remote server to HDFS.

As tuples belonging to a specific key might be arbitrarily distributed in all chunks, we introduce a simple hash function to cluster tuples into buckets according to disjoint key ranges. If we have numeric keys (e.g. id1 for links) and a key range of 1K (0..1K; 1K..2K; ...), the bucket index for each tuple is derived from `b_index=round(id1/1K)`. With bucket index and key range, the coordinator can tell exactly which bucket should be available in HBase to complete the results for an incomplete query. As shown on the left side of Figure 3, *dist_ch* jobs take fix-sized chunks as inputs and generate a set of buckets of dynamic

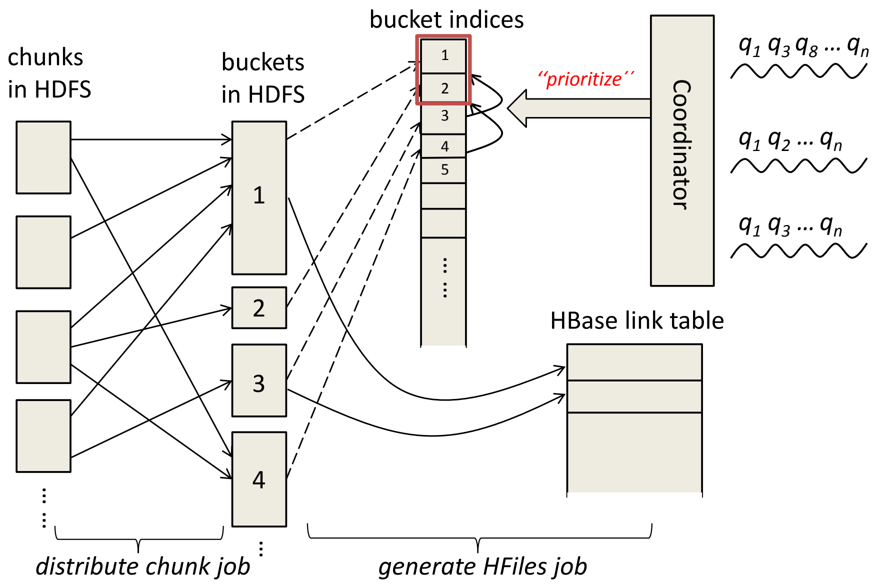

Fig. 3. Offline & Online MapReduce Jobs

sizes based on id1. This can be explained by the LinkBench's observation on the distribution of nodes' outdegrees described in Section 1. Here, a bucket might contain a large number of links which belong to a node with very high outdegree.

3.2 Online HBase Bulk Load and Query Coordination

Once the remote files are copied to local HDFS, our system starts to accept queries from the client side and a *gen_HF* job runs continuously to finish the remaining bulk load work. Three components are involved here: the client-side coordinator, two tables in MySQL (bucket index table and job status table) and the online *gen_HF* job in HBase layer (see Figure 1).

A *gen_HF* job is a MapReduce-based bulk loading job that is executed in several runs in the HBase layer. In each run, it takes HDFS files in "bucket" directories (directed by bucket indices) as input and generates HFiles as results (see Subsection 2.2). Tuples in HFiles can be randomly read without batch processing. The cost of the *gen_HF* job is dominated by sorting. When the local memory on each region server cannot hold the entire set of Put objects for in-memory sorting, expensive disk-based external sorting occurs. Hence, a *gen_HF* job each time will take only the top two buckets (included in the red rectangle in Figure 3) as input to avoid external sorting on local region servers.

The coordinator plays an important role in our demand-driven bulk loading scheme. It maintains a set of four-element tuples (`b_index`, `j_stat`, `k_range`, `q_list`) at runtime. The `b_index` is the bucket index which directs the *gen_HF* job to the input files in this bucket. The `k_range` represents the key range of these input files which will be further checked by an incoming query whether this bucket can contain required tuples. The `b_index` and `k_range` of all the buckets are initially read from the MySQL-side bucket index table at once. Note that, after the hot link data has been bulk loaded into MySQL, a special bucket will be created to contain the `k_range` of MySQL-side "link table".

Furthermore, before the *gen_HF* job starts a run, it registers its two input bucket indices in the job status table in MySQL. When the job is done, it updates its status in the job status table, whose content will be periodically pulled by the coordinator to maintain the `j_stat` elements for all buckets. At the beginning of system uptime, files in most of the buckets have not been transformed to HFiles and thus are not available in HBase. It's much likely that the incoming queries at that moment cannot be completely executed and are further pushed into the query list `q_list` of certain buckets. The coordinator will release the queries in a `q_list` once the `j_stat` states that this bucket is readable. Moreover, the coordinator will also sort the four-element tuples according to the size of `q_list` due to emergency so that the *gen_HF* job will always work on transforming the top two buckets with the largest number of waiting queries.

As an example of demand-driven bulk loading shown in Figure 3, three client-side threads keep sending queries to the coordinator. Most of them contain queries that would access tuples in the key ranges of bucket `1` and `3`. The coordinator checks the size of the query waiting list and prioritizes the *gen_HF* job execution sequence for bucket `1` and `3`. Hence, after the next run, query q_1 and q_3 will be released by the coordinator since the required tuples now can be found in the HBase "link table".

4 Experiments

In Section 2, we mentioned our motivation of partitioning and loading large-scale link sets of a social graph to a hybrid storage system (consisting of a MySQL database and a HBase cluster) based on the observation that only a fraction of links is frequently accessed while most of the data is seldom used. To enable fast availability (i.e. fast load speed) of the entire social graph system, we introduced our "Demand-driven Bulk Loading (DeBL)" scheme in Section 3. In this section, we validate our approach by analyzing the experimental results. The performance difference in terms of load speed and query latency is shown by comparing the results using a single MySQL database, using a single HBase cluster or using our DeBL approach. Our approach serves as a compromise between these two systems and outperforms both of them when loading large-scale social graphs.

We used a logical *link* table with its schema (`id1`, `link_type`, `id2`, ..., `data`) to represent links stored in MySQL, HBase or both systems (as links occupy the largest portion in a graph, we ignored loading graph nodes in our test). The test query is the `get_link_list` operation which performs a short range scan to fetch links with given `id1s` and `link_types` and constitutes 50% of the whole workload. We think that this test setup is general and representative.

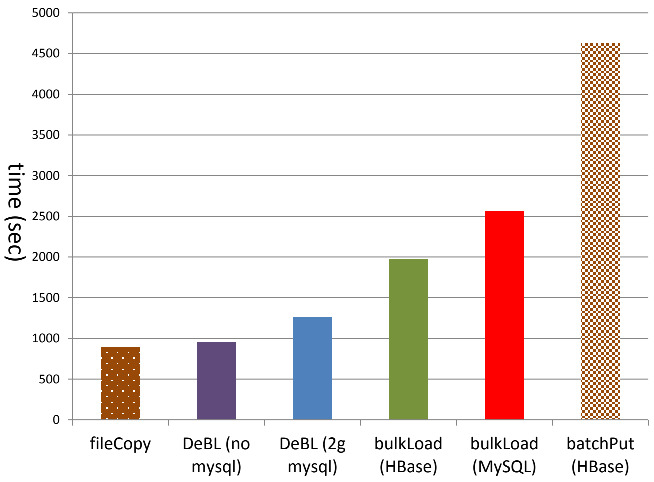

Fig. 4. Loading Time of Different Approaches

(For write operations, both MySQL and HBase can provide fast random write facility. However, we excluded these operations to simplify our test.)

We extended the LinkBench [2] program for our test purpose which is based on a client/server architecture. In the original LinkBench implementation, multiple threads run on the client side to either load links (`load phase`) or send requests (`query phase`) to a server-side "link table" (using MySQL `INNODB` engine) in a MySQL database. To compare the load performance of different approaches, we first recorded the latency of bulk loading a CSV input file (100M links, 10GB) from a remote client (through 100Mb/s Ethernet) into a link table in a MySQL instance running on a single-node machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8GB RAM, 1TB SATA-II disk) using `LOAD DATA INFILE` command (the primary key index was first disabled and re-enabled after the file was loaded) [`bulkLoad (MySQL)`]. Since we were only comparing short range scan performance in the query phase later, a faster "link table" using MySQL `MYISAM` engine instead was used which is optimized for heavy read operations (MySQL's batch insert was excluded in this test as `MYISAM` engine uses table-level locking on tables which is very slow for massive, concurrent inserts).

In the second case, we tested the bulk load performance on a HBase cluster - a 6-node HBase cluster (version 0.94.4) with a master running on the same node as the MySQL instance and 5 region servers (2 Quad-Core Intel Xeon Processor X3440, 4×2.53GHz, 4GB RAM, 1TB SATA-II disk) connected by Gigabit Ethernet. A big MapReduce job (the *gen_HF* job) was implemented to generate HFiles and populate a HBase link table after the same input file was copied from remote to local HDFS [`bulkLoad (HBase)`]. Since HBase also provides high write throughput, we also tested the performance of writing links to HBase in batch using HBase's Put method [`batchPut (HBase)`]. To improve performance, the link table was first pre-split evenly in the cluster according to its key distribution to avoid "hotspots" during loading and querying.

Two variants of the load phases were tested using our DeBL approach. The first variant was composed of bulk loading 2GB, recently changed, remote link subsets into the MySQL table, copying the rest 8GB link files in batch from remote client to HDFS (in our HBase cluster) and meanwhile running multiple small MapReduce jobs (the *dist_ch* jobs) in parallel [`DeBL (2g mysql)`]. Another extreme case was shown by the second variant where no files were loaded into MySQL and the entire input file was copied to HDFS [`DeBL (no mysql)`]. In this case, the "'hotness and coldness" in input data was not pre-defined manually but was captured automatically by our coordinator during *gen_HF* phase according to incoming query distribution. To indicate fast availability of DeBL approach, we attached the time taken to simply copy the test input file to server's local file system [`fileCopy`] to the final results as the bottom line as well.

Table 1. Detailed Latencies (sec) in Load Phase

DeBL (no mysql)		DeBL (2g mysql)		bulkLoad (MySQL)		bulkLoad (HBase)	
chunk load:	24.33	mysql load:	463.26	bulk load:	1674.76	HDFS copy:	895.31
dist_ch job:	24.95	hbase load:	793.98	gen. index:	890.57	gen. HFiles:	1082.81
total:	952.17	total:	1257.23	total:	2565.34	total:	1978.12

The results of load latencies are shown in Figure 4 and detailed in Table 1. The latency of `fileCopy` is the bottom line which is 893 seconds and cannot be improved anymore. The result of `DeBL (no mysql)` is 952.17s and very closed to `fileCopy`'s latency. The input file was transferred chunk by chunk and each chunk has 256MB size. With this chunk size, both chunk load job and *dist_ch* job took similar time (24∼25s). As both jobs ran in parallel, the result of `DeBL (no mysql)` could be derived from the sum of `fileCopy`'s time and the latency of the last *dist_ch* job. It provides the fastest starting time of system uptime with near wire-speed. However, incoming queries still have to wait until their files are available in HBase. Another variant `DeBL (2g mysql)` took a little bit longer for bulk loading 2GB hot data into MySQL (including index construction) which is 463.26s and its total latency is 1257.23s.

Latency gets higher when using traditional bulk loading approaches. Using `bulkLoad (MySQL)`, the `LOAD DATA INFILE` command took 1674.76s while re-enabling primary key index spent 890.57s. Using `bulkLoad (HBase)`, copying remote files to HDFS had the same latency as `fileCopy` and generating HFiles reached similar cost (1082.81s) as MySQL's index construction since both processes required sorting on large input files. However, `bulkLoad (HBase)` is faster than `bulkLoad (MySQL)` since HBase is a distributed system where MapReduce's batch processing feature can be exploited. Apart from this difference, both bulk loading approaches still outperforms HBase's fast writes where some overheads like compaction occurred due to HBase's implementation as mentioned in Subsection 2.2.

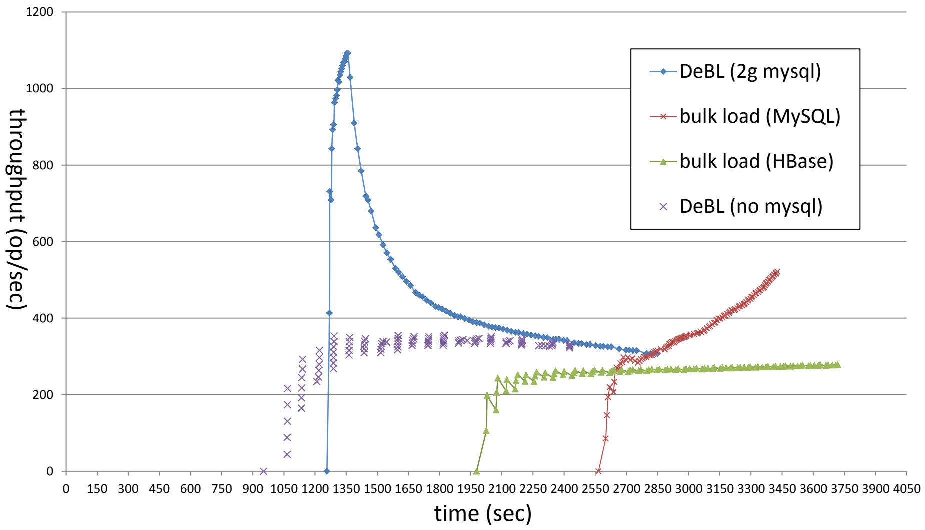

Fig. 5. System Uptime & Query Throughput in four Approaches

The end of load phase indicates the start of system uptime and lower load latency means faster data availability. After the load phase, we ran 50 threads on the client side to continuously send 500,000 *get_link_list* requests to the server. Both MySQL and HBases' bulk loading approaches led to complete data availability and we tracked the query throughputs starting from their system uptime. In this case, the difference of the query throughputs represents the difference of query latencies on MySQL and HBase as well. We tried our best to tune our HBase cluster for example, by enabling block caching, setting up Bloom filters and pre-splitting regions. For 10GB test data, our in-house HBase cluster yet could not cope with a single-node MySQL engine in terms of query throughput as shown in Figure 5. However, as HBase cluster took less time to ingest the input data, its throughput curve started earlier (from 1978.12s, the end of its load phase) to rise and converged at 278.7 op/sec throughput and 3.6 ms query latency in average for each *get_link_list* request. The `bulkLoad (MySQL)` approach took the longest time until all links are available in link table. Its throughput was rising rapidly (till 521.8 op/sec, 1.9 ms) and all the queries were finished in a small time window.

In contrast to traditional bulk loading approaches, our DeBL approach trades complete data availability for fast system uptime. It provides incremental availability to files stored in HDFS on demand. It can be seen in the `DeBL (no mysql)` variant that the system started the earliest at 952.17s but the query throughputs occurred intermittently as relevant file partitions continuously got available in HBase. The throughputs were higher than `bulkLoad (HBase)` at the beginning since less available files needed to be scanned in HBase. Along with growing data size, the throughputs kept close to the highest value in `bulkLoad`

`(HBase)` (332 op/sec). Using `DeBL (2g mysql)`, the system uptime had 300s delay whereas its throughput curve first climbed up drastically and reached its peak 1092.7 op/sec. After that, it began to fall and finally converged with `bulkLoad (HBase)`'s curve. The reason is that a big portion of frequently emerging queries were immediately answered by the 2GB hot data in MySQL at first. As the size of data in MySQL was much smaller, their query latencies were also faster than those on 10GB data in `bulkLoad (MySQL)`. The rest of the queries that could not be answered by MySQL were buffered by the coordinator and released as soon as the data was available in HBase. Important to mention, both DeBL variants were able to digest the entire 500,000 requests before the system uptime began in `bulkLoad (MySQL)`.

5 Related Work

Bulk loading techniques normally serve the loading phase in Extract-Transform-Load (ETL) processes which handle massive data volumes at regular time intervals. A middleware system called "Right-Time ETL (RiTE)" [11] provides ETL processes with INSERT-like data availability, but with bulk-load speeds. Instead of loading entire input data directly into the target table on a server, a server-side, in-memory buffer is used to hold partial rows of this table before they are materialized on disk. Since loading data in memory is much faster, the time window of data loading is shrunk. A logical view is defined to enable query execution on rows stored in both locations. However, problems will occur when large-scale social graphs cannot fit into memory. In our case, we let a distributed file system take over partial load/query jobs on large files from databases.

In [12], Goetz Graefe proposed his idea of fast loads and online B-tree index optimization as a side-effect of query processing. Instead of building complete indexes during data loading, a small, auxiliary structure called *partition filters* (similar to small materialized aggregates [14]) is created for each new loaded partition. With this information, the indexes are optimized incrementally and efficiently on demand according to queries' predicates. This inspired us to use a bucket index table as auxiliary information to identify required file buckets to be available in HBase for incoming queries.

With the advent of "Big Data" and its emerging Hadoop/MapReduce techniques, database vendors now have lots of solutions that integrate open-source Hadoop with their products. IBM's InfoSphere BigInsights and Big SQL [13] is one of them. Big SQL provides a SQL interface to files stored in Hadoop, BigInsights distributed file systems or external relational databases. It allows companies with existing large relational data warehouses to offload "cold" data to cheap Hadoop clusters in a manner that still allows for query access. In this context, our approach exploits the features of underlying MySQL and HBase engines to balance the availability between "hot" and "cold" data.

6 Conclusion

In this work, we first introduced the bulk loading techniques used for MySQL and HBase and then proposed our demand-driven bulk loading scheme. This scheme utilizes a hybrid storage platform consisting of a fast-load/slow-query HBase and a slow-load/fast-query MySQL to accommodate large-scale social graphs, which is a compromise as fast available "hot" graph data and slowly accessible "cold" data. Our experimental results show that our approach provides fast system uptime and incremental availability to "cold" data on demand.

We do not assume that the data partition stored in MySQL is always hot since the "hotness" of files that resides in HBase can still be discovered in our approach. The limitation is that the query latency of HBase is not satisfactory if the data partition stored in HBase gets frequently accessed in the future. Our future work is to remove this limitation by online data re-balancing in the hybrid storage cluster.

References

1. Curtiss, M., Becker, I., Bosman, T., Doroshenko, S., Grijincu, L., Jackson, T., Zhang, N.: Unicorn: a system for searching the social graph. VLDB, 1150–1161 (2013)
2. Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: Linkbench: a database benchmark based on the facebook social graph, pp. 1185–1196. ACM (2013)
3. Borthakur, D., Gray, J., Sarma, J.S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Aiyer, A.: Apache Hadoop goes realtime at Facebook. In: SIGMOD, pp. 1071–1080 (2011)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB, pp. 143–154. ACM (2010)
5. Rabl, T., Gómez-Villamor, S., Sadoghi, M., Muntés-Mulero, V., Jacobsen, H.A., Mankovskii, S.: Solving big data challenges for enterprise application performance management. VLDB, 1724–1735 (2012)
6. Bercken, J., Seeger, B.: An evaluation of generic bulk loading techniques. VLDB, 461–470 (2001)
7. https://dev.mysql.com/doc/refman/5.0/en/insert-speed.html
8. White, T.: Hadoop: The definitive guide. O'Reilly Media, Inc. (2012)
9. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Gruber, R.E.: Bigtable: A distributed storage system for structured data. In: TOCS (2008)
10. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Informatica, 351–385 (1996)
11. Thomsen, C., Pedersen, T.B., Lehner, W.: RiTE: Providing on-demand data for right-time data warehousing. In: ICDE, pp. 456–465 (2008)
12. Graefe, G., Kuno, H.: Fast loads and queries. Transactions on Large-Scale Data-and Knowledge-Centered Systems II, 31–72 (2010)
13. http://www.ibm.com/developerworks/library/bd-bigsql/
14. Moerkotte, G.: Small materialized aggregates: A light weight index structure for data warehousing. VLDB, 476–487 (1998)