HBelt: Integrating an Incremental ETL Pipeline with a Big Data Store for Real-Time Analytics

Weiping Qu(✉), Sahana Shankar, Sandy Ganza, and Stefan Dessloch

Heterogeneous Information Systems Group,
University of Kaiserslautern, Kaiserslautern, Germany
{qu,s_shankar12,s_ganza,dessloch}@informatik.uni-kl.de

Abstract. This paper demonstrates a system called HBelt which tightly integrates a distributed, key-value data store HBase with an extended ETL engine Kettle. The objective is to provide HBase tables with real-time data freshness in an efficient manner. A distributed ETL engine is extended and integrated as an overlay of HBase. Meanwhile, we extend this ETL engine with the capability of processing incremental ETL flows in a pipelined fashion. Delta batches are defined by the MVCC component in HBase to flush the incremental ETL pipeline for multiple concurrent read requests.Experimental results show that high query throughput can be achieved in HBelt for real-time analytics.

1 Introduction

Nowadays, many scalable, distributed data stores have been developed to deliver large scale data analytics over high volume of structured/unstructured data for valuable results. Data is first extracted, transformed and loaded (ETL) from heterogeneous sources into a centralized data store using ETL tools.

In order to meanwhile keep track of updates happening at the sources side, incremental ETL [9,10] has been widely used to propagate only deltas to the analytical systems instead of re-loading source data from scratch. Incremental ETL normally runs the maintenance flows periodically, i.e. hourly, or in micro-batches (minutes). However, for time-critical decision making, it is desirable to have real-time databases which provide queries with up-to-date state of touched tables. This forces ETL engines to propagate deltas to the target system in a very fast pace even with high update ratio in the external sources.

Background and Motivation. In our previous work [1], we introduced a demand-driven bulk loading scheme to allow early uptime for analytical systems by first offloading large amounts of *cold* data into a distributed, scalable, big data store HBase [2]. Data resides in HBase initially and becomes incrementally available in the target system according to the access priorities. Meanwhile, there are more and more updates collected from a variety of external sources. To achieve data freshness for time-critical decision making, an efficient maintenance mechanism is needed to refresh the data that are still buffered in HBase.

T. Morzy et al. (Eds.): ADBIS 2015, LNCS 9282, pp. 123–137, 2015.
DOI: 10.1007/978-3-319-23135-8_9

In this work, we propose our HBelt system which tightly integrates HBase with a pipelined data integration engine extended by an open-source ETL tool (Pentaho Data Integration (Kettle) [3], shortly Kettle) for real-time analytics. HBelt enables HBase tables to keep track of concurrent data changes in external data sources and provides each analytical query with a consistent view of both the base data and the latest deltas preceding the submission of the query. Data changes are propagated to HBase in a query-driven manner. The contributions of this paper are as follows:

– We deploy a Kettle environment directly in the same cluster shared by HBase. A copy of an ETL flow instance runs on each HBase working node. Besides, a HBase-specific partitioner is implemented in Kettle to distribute captured deltas to the correct HBase working nodes.
– We define our consistency model in HBelt and embed the Multi-Version Consistency Control (MVCC) component of HBase into Kettle. The MVCC component is used to define delta batches that need to be propagated to the target HBase tables for answering specific query requests.
– We propose a pipelined Kettle engine to process different delta batches in parallel. Kettle is geared towards data pipelining for high throughput of an ETL flow.

The remainder of this paper is as follows. We relate our work to several recent attractive work in different domains in Sect. 2. We give a brief introduction of key components in HBase and Kettle in Sect. 3. In Sect. 4, we introduce our HBelt system which integrates HBase with Kettle in terms of consistency and performance. Experiments are conducted and discussed in Sect. 5.

2 Related Work

PigLatin [7] is a script language developed in the Pig project. Pig scripts can be used to perform batch ETL jobs that run as MapReduce [8] jobs and thereby can be seen as a distributed ETL engine. Map/Reduce tasks are executed remotely directly over data stored in cluster nodes, thus delivering high scalability and parallelism. Furthermore, Pig also supports loading data into HBase through its pre-defined HBaseStorage class. Regarding function shipping, HBelt is similar to Pig which executes ETL flows directly on remote data nodes. However, HBelt allows each query/request to access up-to-date state of data by integrating MVCC component into Kettle. Meanwhile, we implemented pipelined version of ETL flows to enable HBase to efficiently react to trickle-feeding updates instead of batch processing.

Real-time databases result from the trend of merging OLTP & OLAP workloads, also known as *one-size-fits-all* databases. Hyper [13] is a typical example of these databases and is designed as an in-memory database. In Hyper, updates in OLTP workloads are performed in sequence in a single thread while each OLAP query session will see a snapshot of the current table state in a child thread forked from the parent update thread. Another example related to our work is

R-Store [6] which stores both real-time data and historical cubes in HBase. Historical cubes are used for OLAP queries and get incrementally maintained with the updates captured from real-time OLTP data by a streaming MapReduce called HStreaming. One difference between HBelt and R-Store is the location of OLTP data. Real-time data resides in R-Store while HBelt assumes a more general situation that real-time deltas are captured from external OLTP sources using the extract component in ETL.

Golab et al. proposed temporal consistency and scheduling algorithms in their real-time stream warehouse [11,12]. Each real-time query always accesses the latest value preceding the submission time of the query. In their stream warehouse, data is divided into multiple partitions based on consecutive time windows. Each partition represents data in a certain time window and there are three consistency levels defined for queries, i.e. `open`, `closed` and `complete`. A partition is marked as `open` if data currently exists in or is expected to exist in the partition. From the query perspective, a `closed` partition implies that the scope of pending data has been fixed, whereas data is expected to arrive in a limited time window. This means that the query can be executed over base data that might be incomplete. The `complete` level is the strongest query consistency and all the data has arrived in the partition. We reuse this notion of temporal consistency in our work for consistency control by extending the MVCC component in HBase.

3 Background

In this section, we give a brief introduction of HBase and Kettle as background and describe only the components which are relevant to our work.

3.1 HBase

HBase [2] is a scalable, distributed key-value store that is widely used to deliver real-time access to big data. It follows a master/slave architecture. In HBase, a table is horizontally partitioned into a set of regions with non-overlapping key ranges. Each region contains a set of in-memory key-value lists called *memStore* and multiple on-disk *storeFiles*. Once a memStore fills up, it is flushed onto disk as a new storeFile. All data (regions) reside only in slave nodes called HRegionServers while the master node has only meta-data information which specifies how the regions with different key ranges are partitioned across HRegionServers.

As a data store, it provides only primitive operations (i.e. put, get and scan) based on a given row key. Based on the meta-data information (row key-HRegionServer mappings), a master node delegates all the put/get operations to corresponding HRegionServers where the actual operations take place. For large scale data analytics over HBase, there have already been efforts that implements an extra SQL layer over HBase which accesses tables stored in HBase through these primitive operations [4,5].

In HBase, only two transaction isolation levels are guaranteed, i.e. read uncommitted and read committed. In order to achieve consistency between concurrent reads and writes, a component called Multi-Version Consistency Control (MVCC) is used. Each region contains a MVCC instance which maintains an internal *write queue*. A write queue is a list of Write Entry (*we*) elements which is used to assign a unique write number to an individual write or a batch of writes. Writes are not allowed to commit until their preceding writes have committed in this write queue. In this way, sequential writes are guaranteed in HBase. When a get/scan operation is issued with read committed as the transaction isolation level, the MVCC component returns the latest committed write number to this thread as read point *readPt* for fetching key-values whose write numbers are lower than or equal to this value in this region.

3.2 Kettle

Kettle [3] (or PDI) is an open-source ETL tool that has been widely used in the research community and provides a full-fledged set of transformation operations (called *step* in Kettle). A stream or batch of files are taken as input and processed row by row in each step. During flow execution, each step is running as an individual thread. Step threads are connected with each other through an in-memory queue called *RowSet*. The results of a preceding step are put in its output rowset which in turn is the input rowset of its subsequent step where rows get fetched. Step threads kill themselves once they are finished with their batch of files.

Kettle also enables a cluster execution mode in which multiple copies of the same flow instance can run in parallel over distributed nodes for better performance. The cluster environment follows a master/slave architecture. The input files of the flows running on the slave nodes are constructed by partitioning and distributing rows in master node according to a user-defined partition schema.

4 HBelt System

In this Section, we introduce our HBelt system, which integrates a distributed, HBase big data store with an extended, pipelined data integration engine based on Kettle for real-time analytics. Analytical queries are issued to a relational database layer over HBase in which actual target tables reside. In order to keep track of concurrent data changes at the source side, the internal consistency in HBase is maintained by multiple Kettle pipeline instances before each query is executed. A single query sees a consistent view which consists of the base data and the latest deltas preceding the submission time of this query. Furthermore, we try to reduce the synchronization delay by introducing two kinds of parallel computing techniques: data partitioning and data pipelining. Therefore, the objective of HBelt is to ensure both consistency and performance. The architecture is illustrated in Fig. 1.

4.1 Architecture Overview

As described in Sect. 3, a table stored in HBase are horizontally partitioned to a set of regions with non-overlapping key ranges and distributed over multiple HRegionServers. Current Kettle implementation (since Version 5.1) has provided a so-called HBase Output step to maintain a HBase table by using a single flow instance. All calculated deltas have to go through this step to arrive in target HRegionServers. However, since both HBase and Kettle follow the master/slave architecture, it is desirable to utilize the essence of distributed processing from both systems in terms of integration. In HBelt, the same number of the flow instance copies are constructed as the number of HRegionServers and further executed directly on each single HRegionServer node.

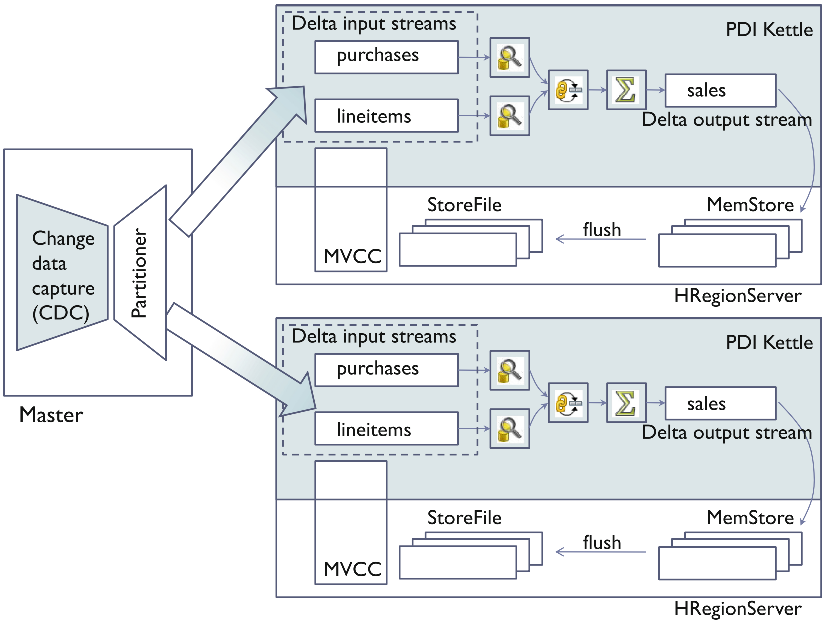

Fig. 1. HBelt architecture

Take a logical ETL flow as an example, which processes data changes captured from external *purchases* and *lineitems* sources to maintain the target *sales* table in HBase. In the master node (at the left side of Fig. 1), a change data capture (CDC) step uses methods like log-sniffing [14] or timestamps to capture the source deltas. In order to forward source deltas to the right HRegionServers for further flow execution, both the keys in the deltas and the key ranges of regions stored in HBase tables need to be considered. This is done by a component called *Partitioner*. In this example, *purchase* rows have *purc_id* as key and both *lineitems* rows and the *sales* table have compound keys (*purc_id*, *item_id*). The partitioner component fetches cached meta-data of the *sales* table from HBase in the same master node and forms a user-defined partition schema in Kettle. This meta-data shows the mapping from row keys to HRegionServers, based on which the *lineitems* deltas can be distributed to server nodes correctly. For a *purchases* row whose *purc_id* might span across regions in multiple HRegionServers, copies of this *purchases* row are sent to HRegionServers along with *lineitems*. In this way, we guarantee that calculated deltas for the target *sales* table should reside on the correct HRegionServer.

So far, we have introduced a sub-flow which consists of two steps: CDC and Partitioner. This sub-flow runs independently of query requests on HBase tables and feeds source deltas continuously to the delta input streams in HRegionServers to reflect the concurrent updates on the source side.

4.2 Consistency Model

In this subsection, we define our consistency model in HBelt for real-time analytics over HBase. Take an example shown in Fig. 2. At the upper left side, there is a traditional transaction log file recording five transactions ($T_1 \sim T_5$) committed from t_1 to t_5, respectively. The CDC process mentioned in previous subsection is continuously extracting these changes from the log file and sending corresponding deltas to the delta input streams of both of the Kettle flow instances (in this case only two flow instance copies are running on two individual HRegionServers).

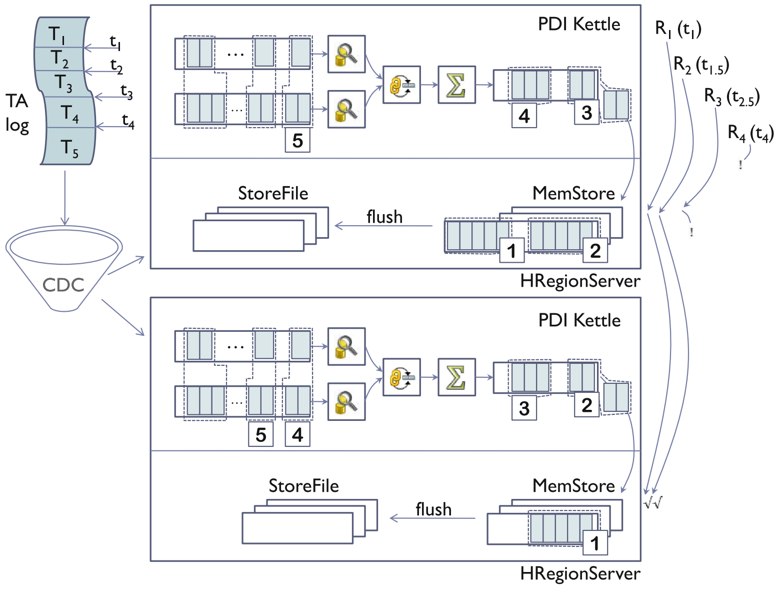

Fig. 2. Consistency model

Meanwhile, four distinct requests have been issued to HBase to perform scan operations over regions stored in these two HRegionServers. The first scan request R_1 occurs at timestamp t_1 which forces HBelt to refresh existing HBase table using changes (e.g. insertions, updates and deletions) derived from the first transaction T_1 which is committed at t_1. Once these changes have been successfully propagated and stored into the memStores in these two HRegionServers, R_1 is triggered to started immediately. Although the second scan request R_2 is issued at later time $t_{1.5}$, it still precedes the committing time of the second transaction T_2 (at t_2). Hence, it shares the same state of the HBase table as R_1. The third scan request R_3 has its occurring time $t_{2.5}$ which succeeds the committing time of T_2 and precedes the committing time of T_3. Since the deltas from T_2 are only available in the first HRegionServer, R_3 first completes the scan operation over regions in the first HRegionServer and waits for the regions in the second HRegionServer to be refreshed by T_2's committed changes. To answer the fourth request R_4, relevant regions stored in both HRegionServers need to be upgraded by the deltas from T_1 to T_4. Since neither of Kettle flows has finished propagated these deltas to HBase, R_4 is suspended until the HBase table is refreshed with correct deltas.

4.3 MVCC Integration for Delta Batches

In this subsection, we show how maintenance flows and query requests are scheduled in each HRegionServer to achieve the consistency we defined in previous subsection. Recall that in HBase the consistency in each region is maintained by a Multi-Version Consistency Control instance (see Sect. 3) where a local *write queue* is used to ensure sequential writes. A write queue maintains a list of `open` Write Entries *we* for assigning unique write numbers to batches of writes during insertions. Writes are only visible after they are committed and corresponding *we*s are marked as `complete`. However, in order to make each query request see a consistent view of base data and deltas, the current MVCC implementation in HBase has to be extended to meet our needs.

At any time, there is always one and only one `open` write entry *we* in the write queue. While source deltas continuously arrive in each HRegionServer, instead of triggering the maintenance flow to start immediately, deltas are first buffered in input streams and all of them are assigned the write number of this open *we*. We define that all the deltas sharing the same write number belong to a delta batch with a batch id. Once a read request is issued by an analytical query, this *we* is first marked as `closed` instead of `complete` (Here we embedded the temporal consistency described in Sect. 2 in our work). The `closed` state indicates that the maintenance flow now gets started to digest this delta batch with *we*'s write number as batch id and the final calculated deltas with this batch id have not yet completely arrived in HBase. Therefore, the read request awaits the completion of its maintenance flow and gets pushed into a waiting list *read queue*. Meanwhile, a new write entry we' is created and inserted into write queue to *paint* newly incoming deltas with we''s write number.

At the time the last row with (*we*'s) batch id gets successfully inserted into HBase memStore by the final maintenance step, *we* is finally marked as `complete` and gets removed. All waiting reads in the *read queue* are notified of this event and check whether the complete batch id matches their local ones. The read request which waits for exactly this event gets started to continue with either a get or scan operation. Even though during the scan operation more new delta batches are inserted into the same regions, this read request would not be interfered with since it has an older batch id which restricts the access of rows with newer batch ids. In this way, we guarantee that each read request always sees the latest value of a consistent view of base data and deltas preceding its submission time.

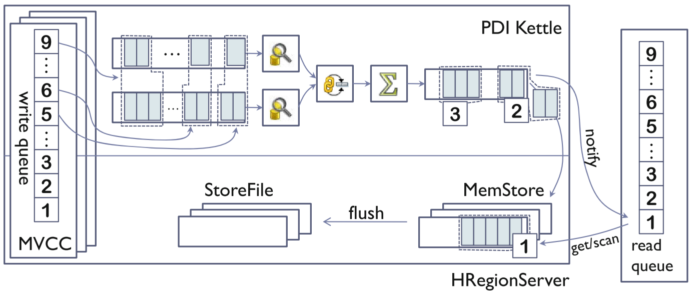

Fig. 3. MVCC integration for delta batches on HRegionServer

Figure 3 illustrates a snapshot taken at the time nine read requests have been issued by analytical queries. The arrival of these requests forces MVCC to group the corresponding deltas into nine batches which were once buffered in the streams before each occurrence of request. These read requests are waiting in the read queue until their delta batches get finished through the maintenance flow. Meanwhile, nine pending batches are denoted by the write entries stored in the write queue of MVCC component. They are all marked as `closed` except the first one since the first delta batch has been successfully moved to HBase memStore and can be made accessible to the first request. Thus, the first request is reactivated by the final maintenance step and continues with the get/scan operation. The second and third batches have already been put into the output streams and their requests are about to start. Note that, due to high request rate, delta batches 5–9 are still buffered in the input streams since the maintenance flow is still processing previous batches.

4.4 Pipelining Delta Batches in Kettle

As we can see from the previous subsection, the maintenance flow could be busy with processing different batches issued by multiple requests, especially with a high request rate. Hence, there is a need to speed up the performance of the maintenance flow. For each read request, in order to keep track of concurrent updates at the source side, the synchronization latency incurred by the maintenance flow is fixed. However, another potential optimization opportunity is to increase the throughput of the system. To address this, a pipelined flow engine based on Kettle is proposed.

As described in Sect. 3, the original Kettle implementation simply takes a stream/batch of data as input with no comprehension of different consecutive batches. It is important to distinguish different batches for specific transformation operations e.g. sort, aggregation, etc. in our work. Otherwise a maintenance flow could generate incorrect deltas for each read request, leading to inconsistent analytical results. For example, if a sort operation would receive rows from two delta batches and process them at the same time, the results coming out of this operation would be totally different from the results of sorting two batches separately. This also holds for aggregation operations like sum() or avg().

Algorithm 1. Step Implementation in Pipelined Kettle

```
Input: rq // read queue which bufferes waiting read requests.
       in // intput rowsets
       out // output rowsets
Init:  readPt // local read point
       index // index used to iterate read queue.
1  while true do
2      if rq is empty || in is empty then
3          wait();
4      readPt ← rq[index++];
5      init(); // clear local caches, counters, etc.
6      while in.getRow().batchID == readPt do
7          r ←processRow();
8          out.add(r);
9          out.notify();
10     depose();
```

In this work, we extended Kettle to a pipeline flow engine which is able to react to different mini-batch jobs at the same time while still guaranteeing consistency. The extension of a single step thread is given above (see Algorithm 1). All steps in the maintenance flow share the same *read queue* which holds a list of pending read requests mentioned in previous subsection. Furthermore, each step maintains a local index which points at certain read request in the queue as a local read point *readPt*. This *readPt* is actually the batch id of the delta batch that needs to be processed. Once a step successfully fetches a batch id that matches the id of the rows in its input rowset, this step first initializes itself by clearing local caches and counters. After a row is processed, in addition to putting the result into the output rowset, it notifies its subsequent step of the existence of the output. When the batch is finished, instead of killing itself

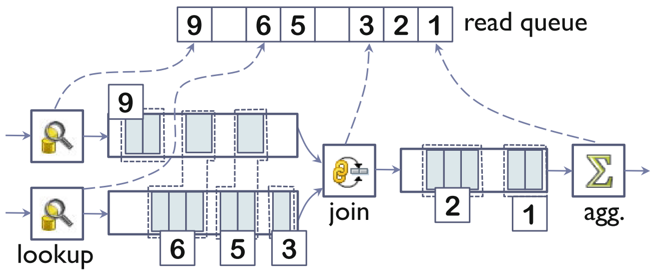

Fig. 4. Pipelined Kettle

as in the original implementation, it deposes itself (e.g. release used database connections) and tries to fetch the next read request in the queue.

As shown in Fig. 4, a pipelined Kettle flow is being flushed by nine delta batches. Due to diverse operational costs, the lookup step in the upper branch of the join step has already started to work on the ninth batch while another lookup step in the lower branch is still working on the sixth one. However, the join step would not continue with processing the rows in subsequent (e.g. fifth or sixth) batches until it makes sure that there is no more row of batch id 3 existing in neither of its input rowsets. Even though the fifth and sixth batches are already available, they are still invisible to the join step since the current *readPt* is still three. Data pipelining is introduced here to increase the throughput of the maintenance flow. However, the synchronization latency for each request is not improved or sometimes even increased, for example, the fifth batch cannot start until the join step finished with all the deltas in the third batch. We will examine it in the experiments.

5 Experimental Results

The objective of HBelt is to provide get/scan operations in HBase with real-time data access to the latest version of HBase's tables by tightly integrating an ETL engine, i.e. Kettle, with HBase. Though current Kettle (since Version 5.1) has implemented "HBase Output" step towards Big Data Integration, in our scenario, sequential execution of a single Kettle flow at once to maintain target HBase tables for time-critical analytics could lead to long data maintenance delay at high request rate. In this section, we show the advantages of our HBelt system by comparing its performance in terms of maintenance latency and request throughput with the sequential execution mode. We mainly examine the performance improvements by using data partitioning and data pipelining techniques in HBelt.

In the experiments, our HBelt ran on a 6-node cluster where a node (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8 GB RAM, 1TB SATA-II disk) served as the master and the rest five nodes (2 Quad-Core Intel Xeon Processor X3440, 4×2.53 GHz, 4 GB RAM, 1TB SATA-II disk, Gigabit Ethernet) were the slave nodes running HRegionServer and Kettle threads (see Subsect. 4.1). Meanwhile, the same cluster was used to accommodate an original version (0.94.4) of HBase connected with a Kettle engine (Version 5.1) running on a client node (Intel Core i7–4600U Processor, 2×2.10 GHz, 12 GB RAM, 500GB SATA-II disk) to simulate the sequential execution mode.

We used TPC-DS benchmark [15] in our test. A *store_sales* table (with SF 10) resided in HBase and was maintained by a Kettle flow with the update files *purchases* (♯: 10K) and *lineitems* (♯: 100K) generated by TPC-DS *dsdgen*. The maintenance flow is depicted in Fig. 5. Purchases and lineitems are the delta files and are joined together in an incremental fashion after applying several surrogate key lookup steps. The intermediate join results are further aggregated as the final delta rows for the target store sales table. In sequential execution mode, the source delta files (purchases & lineitems) resided in the client node and were used as input for the Kettle flow to populate the store sales table in the 6-node HBase cluster using HBase Output. However, in HBelt mode, these source delta files were initially stored in the master node and later continuously distributed and fed to the five slave nodes where two input rowsets were used to buffer delta rows as delta input streams (instead of CSV Input steps). Furthermore, in contrast to sequential execution mode, each region was the target output instead of "HBase Output" step.

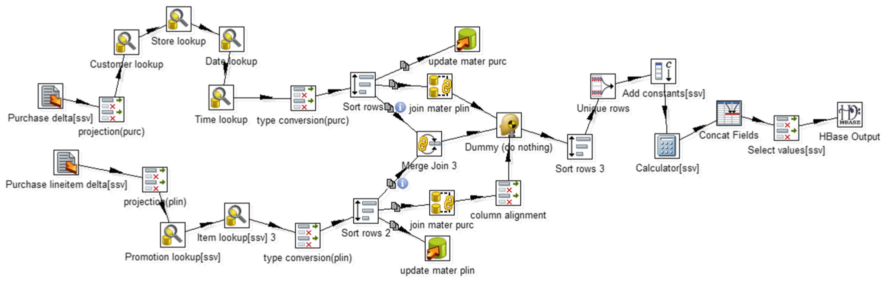

Fig. 5. Test maintenance flow in kettle

Data Pipelining: We first examined the performance improvement associated with the data pipelining technique implemented in the Pipelined Kettle component of our HBelt. The *store sales* table was not split in HBase and had only one region. Thus, only one HRegionServer was activated to serve issued request load and only one pipelined Kettle instance was dedicated to refresh the target table with *purchases* and *lineitems* delta files. Moreover, the delta files were split evenly to 210 chunks to emulate the input deltas to maintain the target table for 210 read requests occurring consecutively in a small time window.

The maintenance latency for each request is shown in Fig. 6. In sequential execution mode (SEQ), the same Kettle flow ran 210 times at the client side one flow at once to refresh target HBase table with 210 delta chunks. The latency difference between two adjacent requests is the duration of one flow execution.

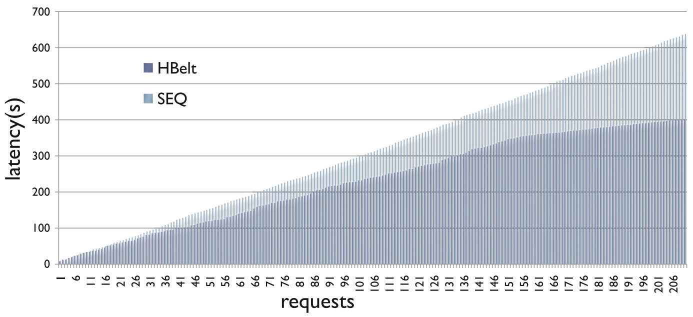

Fig. 6. Maintenance latencies of 210 consecutive read requests on single node

Since each flow execution took the same size of delta chunk as input, the maintenance latency grows linearly. The last request has to wait for the completions of preceding 210 flow executions (∼10.5 min). Using HBelt, the flow pipeline shown in Fig. 5 was flushed by 210 delta batches at the same time. The latency difference between two adjacent requests depends on the slowest step in the pipeline rather than one complete flow execution. In summary, HBelt outperforms SEQ in terms of maintenance latency even though only one region existed in the HBase cluster, i.e. no data partitioning parallelism. Each request started earlier than in SEQ. The synchronization delay for the last request is 400 s, thus increasing the performance by ∼30 %. This proves that HBelt is able to deliver high throughput at a high request rate or in case of "hotspot" issue in HBase, i.e. a single HRegionServer has a higher load than others.

Data Partitioning: We show another advantage of HBelt here: running one pipelined Kettle instance directly on each individual HRegionServer. Firstly, the *store sales* tables were evenly pre-split to 10 regions with non-overlapping row key ranges over 5 HRegionServers, thus each HRegionServer was active and managed 2 regions. Secondly, the request load consisted of a thousand scan operations in which each individual Region[1→10] was scanned by 50 scan operations, subsequent 100 operations scanned Regions (1∼3), 100 operations scanned Regions (4∼6), 100 operations scanned Regions (6∼8), 100 operations scanned Regions (8∼10) and the rest 100 operations scanned the entire table. Hence, each request required in average only 2/7 portion of the table to become up-to-date before it was executed. Finally, we generated a set of delta files *purchases* and *lineitems* of ten sizes {♯: (10 K & 120 K), (20 K & 240 K), ..., (100 K & 1200 K)} each of which were further split to 1000 chunks to simulate the delta inputs for the 1000 scan requests. In each chunk only 2/7 portion in average is needed to refresh the necessary regions for one request.

The request throughputs with different delta size settings are shown in Fig. 7. As the baseline, the request throughput in SEQ decreases steadily from 2.78 (♯requests/s) to 0.46 (♯requests/s) with increasing delta sizes, which indicates growing maintenance overhead. The throughput in SEQ mode is much lower than that in HBelt since two scan operations have to be executed sequentially

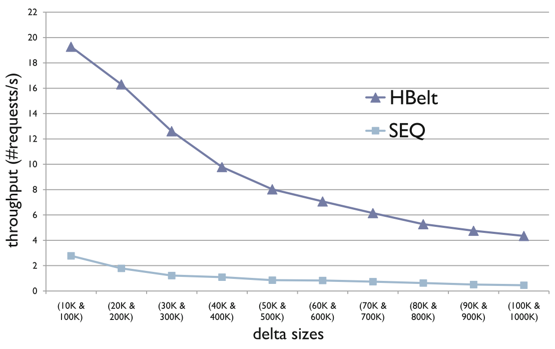

Fig. 7. Request throughput after issuing 1000 requests using diverse delta sizes

no matter how many deltas are really needed to answer certain request. HBelt provides much higher throughput (19.28 to 4.35 ♯requests/s). The efficiency is two fold. Due to data partitioning, HBelt is able to propagate deltas for concurrent requests with non-overlapping key ranges at the same time. For example, a scan operation which accesses Region(1∼3) has no conflict with another scan operation which touches Region(4∼6). Separate ETL pipeline can refresh independent regions at the same time. Meanwhile, since deltas were split and distributed over multiple ETL pipeline instances, the size of input deltas dropped drastically and the latency became less as well. In addition to data partitioning, pipelined Kettle still provides data pipelining parallelism for multiple concurrent requests arriving at the same HRegionServer.

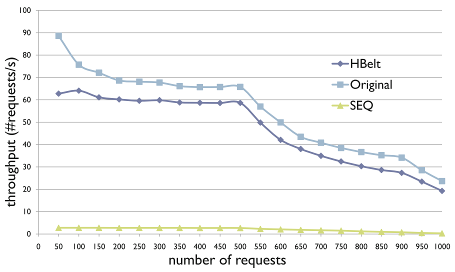

Fig. 8. Request throughput with small deltas (10 K purchases & 100 K lineitems)

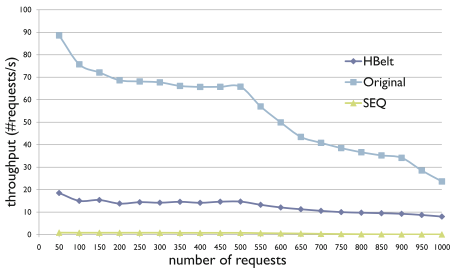

Fig. 9. Request throughput with large deltas (50 K purchases & 500 K lineitems)

Figures 8 and 9 compare the throughput with increasing requests among three settings: HBelt, sequential execution mode and an original HBase setting which does not have maintenance overhead incurred by our ETL pipelines. With small delta sizes (10 K purchases & 100 K lineitems), HBelt achieves performance much similar to original HBase which does not guarantee data freshnees. However, as the size of delta grows, the request throughput of HBelt dropped significantly while it still outperforms the sequential execution mode.

6 Conclusion

In this work, we introduced our HBelt system which integrates an ETL engine Kettle with a big data store HBase to achieve real-time analytics over tables stored in HBase. The integration utilized the architectural essence of both systems, i.e. master/slave architecture. A copy of the Kettle flow instance runs directly on each HBase data node. File inputs are partitioned using our HBase-specific partitioner and further distributed over these data nodes, thus allowing multiple Kettle flow instances to work synchronously for concurrent non-conflicting requests. In this way, we provide data partitioning parallelism in HBelt. Furthermore, we defined the notion of our consistency model to enable each request to see the latest version of tables preceding the request submission time. The consistency component in HBase is embedded in Kettle to identify correct delta batches for answering specific HBase requests. Moreover, we extended Kettle to a pipelined version which is able to work on multiple distinct delta batches at the same time. A pipelined Kettle flow can be flushed by a large number of delta batches, thus increasing request throughput. Finally, the experimental results show that HBelt is able to reduce maintenance overhead and raise request throughput for real-time analytics in HBase.

References

1. Qu, W., Dessloch, S.: A demand-driven bulk loading scheme for large-scale social graphs. In: Manolopoulos, Y., Trajcevski, G., Kon-Popovska, M. (eds.) ADBIS 2014. LNCS, vol. 8716, pp. 139–152. Springer, Heidelberg (2014)
2. http://hbase.apache.org
3. Casters, M., Bouman, R., Van Dongen, J.: Pentaho Kettle Solutions: Building Open Source ETL Solutions with Pentaho Data Integration. John Wiley & Sons, Indianapolis (2010)
4. https://wiki.trafodion.org/
5. http://phoenix.apache.org/
6. Li, F., Ozsu, M.T., Chen, G., Ooi, B.C.: R-Store: a scalable distributed system for supporting real-time analytics. In: IEEE 30th International Conference on Data Engineering, ICDE 2014, pp. 40–51. IEEE, March 2014
7. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1099–1110. ACM, June 2008
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
9. Vassiliadis, P., Simitsis, A.: Near real time ETL. In: Kozielski, S., Wrembel, R. (eds.) New Trends in Data Warehousing and Data Analysis. AIS, pp. 1–31. Springer, Cambridge (2009)

10. Jörg, T., Dessloch, S.: Near real-time data warehousing using state-of-the-art ETL tools. In: Castellanos, M., Dayal, U., Miller, R.J. (eds.) BIRTE 2009. LNBIP, vol. 41, pp. 100–117. Springer, Heidelberg (2010)
11. Golab, L., Johnson, T., Shkapenyuk, V.: Scheduling updates in a real-time stream warehouse. In: IEEE 25th International Conference on Data Engineering, ICDE 2009, pp. 1207–1210. IEEE, March 2009
12. Golab, L., Johnson, T.: Consistency in a stream warehouse. In: CIDR, Vol. 11, pp. 114–122 (2011)
13. Kemper, A., Neumann, T.: HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: IEEE 27th International Conference on Data Engineering, ICDE 2011, pp. 195–206. IEEE, April 2011
14. Kimball, R., Caserta, J.: The Data Warehouse ETL Toolkit. John Wiley & Sons, Indianapolis (2004)
15. http://www.tpc.org/tpcds/