Real-Time Snapshot Maintenance with Incremental ETL Pipelines in Data Warehouses

Weiping Qu(✉), Vinanthi Basavaraj, Sahana Shankar, and Stefan Dessloch

Heterogeneous Information Systems Group,
University of Kaiserslautern, Kaiserslautern, Germany
{qu,k_vinanthi12,s_shankar,dessloch}@informatik.uni-kl.de

Abstract. Multi-version concurrency control method has nowadays been widely used in data warehouses to provide OLAP queries and ETL maintenance flows with concurrent access. A snapshot is taken on existing warehouse tables to answer a certain query independently of concurrent updates. In this work, we extend this snapshot with the deltas which reside at the source side of ETL flows. Before answering a query, relevant tables are first refreshed with the exact source deltas which are captured at the time this query arrives (so-called query-driven policy). Snapshot maintenance is done by an incremental recomputation pipeline which is flushed by a set of consecutive deltas belonging to a sequence of incoming queries. A workload scheduler is thereby used to achieve a serializable schedule of concurrent maintenance tasks and OLAP queries. Performance has been examined by using read-/update-heavy workloads.

1 Introduction

Nowadays companies are emphasizing the importance of data freshness of analytical results. One promising solution is executing both OLTP and OLAP workloads in an 1-tier *one-size-fits-all* database such as Hyper [8], where operational data and historical data reside in the same system.Another appealing approach used in a common 2-tier or 3-tier configuration is near real-time ETL [1] by which data changes from transactions in OLTP systems are extracted, transformed and loaded into the target warehouse in a small time window (five to fifteen minutes) rather than during off-peak hours.Deltas are captured by using Change-Data-Capture (CDC) methods (e.g. log-sniffing or timestamp [9]) and propagated using incremental recomputation techniques in micro-batches.

Data maintenance flows run concurrently with OLAP queries in near real-time ETL, in which an intermediate Data Processing Area (DPA, as counterpart of the data staging area in traditional ETL) is used to alleviate the possible overload of the sources and the warehouse. It is desirable for the DPA to relieve *traffic jams* at a high update-arrival rate and meanwhile at a very high query rate alleviate the burden of locking due to concurrent read/write accesses to shared data partitions. Alternatively, many databases like PostgresSQL [12] use

S. Madria and T. Hara (Eds.): DaWaK 2015, LNCS 9263, pp. 217–228, 2015.
DOI: 10.1007/978-3-319-22729-0_17

a Multi-Version Concurrency Control (MVCC) mechanism to solve concurrency issues. If serializable snapshot isolation is selected, a snapshot is taken at the beginning of a query and used during the entire query lifetime without interventions incurred by concurrent updates.However, for time-critical decision making, the snapshot should contain not only base data but also the latest deltas preceding the submission of the query [7]. The scope of our work is depicted in Fig. 1. Relevant warehouse tables are first updated by ETL flows using deltas with timestamps smaller or equal to the query submission time. A synchronization delay is thereby incurred by a maintenance flow execution before a query starts. We assume that a CDC process runs continuously and always pulls up-to-date changes without maintenance anomalies addressed in [5].Correct and complete sets of delta rows arrive in the DPA and are buffered as delta input streams. The submission of a query triggers a maintenance transaction to refresh relevant tables using a batch of newly buffered deltas (called query-driven refresh policy). A delta stream occurring in a specific time window is split into several batches to maintain snapshots for a number of incoming queries.

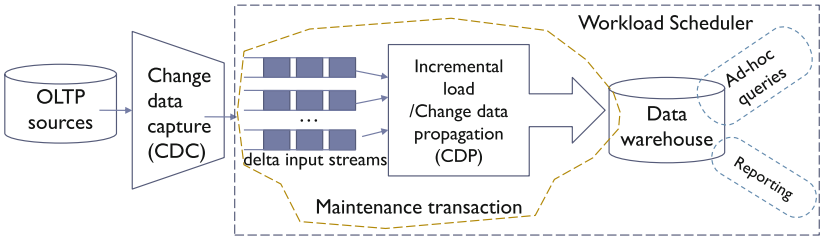

Fig. 1. Consistency scope in warehouse model

In our work, a snapshot is considered as *consistent* if it is refreshed by preceding delta batches before query arrival and is not interfered by fast committed succeeding batches (non-repeatable read/phantom read anomalies). While timestamps used to extend both delta rows and target data partitions could be a possible solution to ensure query consistency, this will result in high storage and processing overheads. A more promising alternative is to introduce a mechanism to schedule the execution sequence of maintenance transactions and OLAP queries. Moreover, sequential execution of ETL flows for maintaining snapshots can lead to high synchronization delay at a high query rate. Therefore, performance needs to be of concern as a main aspect as well.

In this work, we address the real-time snapshot maintenance problem in MVCC-supported data warehouse systems using near real-time ETL techniques. The objective of this work is to achieve high throughput at a high query rate and meanwhile ensure the serializability property among concurrent maintenance transactions on ETL tools and OLAP queries in warehouses. The contributions of this work are as follows: We define the consistency notion in our real-time ETL model based on which a workload scheduler is proposed for a serializable schedule of concurrent maintenance flows and queries that avoids using timestamp-based

approach. An internal queue is used to ensure consistency with correct execution sequence. Based on the infrastructure introduced for near real-time ETL [1], we applied a pipelining technique using an open-source ETL tool called Pentaho Data Integration (Kettle) (shortly Kettle) [10]. Our approach is geared towards an incremental ETL pipeline flushed by multiple distinct delta batches for high query throughput. We define the delta batches as inputs for our incremental ETL pipeline with respect to query arrivals. Three levels of temporal consistency (`open`, `closed` and `complete`) are re-used here to coordinate the executions of pipeline stages. The experimental results show that our approach achieves nearly similar performance as in near real-time ETL while the query consistency is still guaranteed.

The remainder of this paper is constructed as follows. We discuss several related work in Sect. 2. Our consistency model is defined in Sect. 3. A workload scheduler is introduced in Sect. 4 to achieve the serializability property and an incremental recomputation pipeline based on Kettle is described in Sect. 5 for high throughput. We validate our approach with read-/update-heavy workloads and the experimental results are discussed in Sect. 6.

2 Related Work

Thomsen et al. addressed on-demand, fast data availability in so-called right-time DWs [4]. Rows are buffered at the ETL producer side and flushed to DW-side, in-memory tables with bulk-load insert speeds using different (e.g. immediate, deferred, periodic) polices. Timestamp-based approach was used to ensure accuracy of read data while in our work we used an internal queue to schedule the workloads for our consistency model. Besides, we also focused on improving throughput by extending an ETL tool.

Near real-time data warehousing was previously referred to as active data warehousing [2]. Generally, incremental ETL flows are executed concurrently with OLAP queries in a small time window. In [1], Vassiliadis et al. detailed a uniform architecture and infrastructure for near real-time ETL. Furthermore, in [3], performance optimization of incremental recomputations was addressed in near real-time data warehousing. In our experiments, we compare general near real-time ETL approach with our work which additionally guarantees the query consistency.

In [6,7], Golab et al. proposed temporal consistency in a real-time stream warehouse. In a certain time window, three levels of query consistency regarding a certain data partition in warehouse are defined, i.e. `open`, `closed` and `complete`, each which becomes gradually stronger. As defined, the status of a data partition is referred to `open` for a query if data exist or might exist in it. A partition at the level of `closed` means that the scope of updates to partition has been fixed even though they haven't arrived completely. The strongest level `complete` contains `closed` and meanwhile all expected data have arrived. We leverage these definitions of temporal consistency levels in our work.

3 Consistency Model

In this section, we introduce the notion of consistency which our work is building on using an example depicted in Fig. 2. A CDC process is continuously running and sending captured deltas from OLTP sources (e.g. transaction log) to the ETL maintenance flows which propagate updates to warehouse tables on which OLAP queries are executed. In our example, the CDC process has successfully extracted delta rows of three committed transactions T_1, T_2 and T_3 from the transaction log files and buffered them in the data processing area (DPA) of the ETL maintenance flows. The first query Q_1 occurs at warehouse side at time t_2. Before it is executed, its relevant warehouse tables are first updated by maintenance flows using available captured deltas of T_1 and T_2 which are committed before t_2. The execution of the second query Q_2 (at t_3) forces the warehouse table state to be upgraded with additional deltas committed by T_3. Note that, due to serializable snapshot isolation mechanism, the execution of Q_1 always uses the same snapshot that is taken from the warehouse tables refreshed with deltas of T_1 and T_2, and will not be affected by the new state that is demanded by Q_2. The third query Q_3 occurs at $t_{3,5}$ preceding the committing time of T_4. Therefore, no additional deltas needs to be propagated for answering Q_3 and it shares the same snapshot with Q_2.

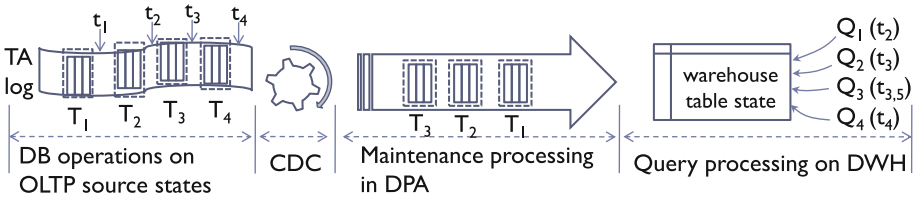

Fig. 2. Consistency model example

In our work, we assume that the CDC is always capable of delivering up-to-date changes to DPA for real-time analytics. However, this assumption normally does not hold in reality and maintenance anomalies might occur in this situation as addressed by Zhuge et al. [5]. In Fig. 2, there is a CDC delay between the recording time of T_4's updates in the transaction log and their occurrence time in the DPA of the ETL flow. The occurrence of the fourth query Q_4 arriving at t_4 requires a new warehouse state updated by the deltas of T_4 which are still not available in the DPA. We provide two realistic options here to compensate for current CDC implementations. The first option is to relax the query consistency of Q_4 and let it share the same snapshot with Q_2 and Q_3. The OLAP queries can tolerate small delays in updates and a "tolerance window" can be set (e.g., 30 s or 2 min) to allow scheduling the query without having to wait for all updates to arrive. This tolerance window could be set arbitrarily. Another option is to force maintenance processing to hang on until CDC has successfully delivered all required changes to DPA with known scope of input deltas for answering Q_4.

With these two options, we continue with introducing our workload scheduler and incremental ETL pipeline based on the scope of our work depicted in Fig. 1.

4 Workload Scheduler

In this section, we focus on our workload scheduler which is used to orchestrate the execution of ETL maintenance flows and OLAP queries. An OLAP query Q_i is first routed to the workload scheduler, which immediately triggers the creation of a new maintenance transaction M_i performing a single run of ETL maintenance flow to update the warehouse state for Q_i. The execution of Q_i stalls until M_i is completed with its commit action (denoted as $c(M_i)$). Query Q_i is later executed in a transaction as well in which the begin action (denoted as $b(Q_i)$) takes a snapshot of the new warehouse state derived by M_i. Therefore, the first integrity constraint enforced by our workload scheduler is $t(c(M_i)) < t(b(Q_i))$ which means that M_i should be committed before Q_i starts.

With arrivals of a sequence of queries $\{Q_i, Q_{i+1}, Q_{i+2}, ...\}$, a sequence of corresponding maintenance transactions $\{M_i, M_{i+1}, M_{i+2}, ...\}$ are constructed. Note that, once the $b(Q_i)$ successfully happens, the query Q_i does not block its successive maintenance transaction M_{i+1} for consistency control since the snapshot taken for Q_i is not interfered by M_{i+1}. Hence, $\{M_i, M_{i+1}, M_{i+2}, ...\}$ are running one after another while each $b(Q_i)$ is aligned with the end of its corresponding $c(M_i)$ into $\{c(M_i), b(Q_i), c(M_{i+1}), ...\}$. However, with the first constraint the serializability property is still not guaranteed since the commit action $c(M_{i+1})$ of a subsequent maintenance transaction might precede the begin action $b(Q_i)$ of its preceding query. For example, after M_i is committed, the following M_{i+1} might be executed and committed so fast that Q_i has not yet issued the begin action. The snapshot now taken for Q_i includes rows updated by deltas occurring later than Q_i's submission time, which incurs non-repeatable/phantom read anomalies. In order to avoid these issues, the second integrity constraint is $t(b(Q_i)) < t(c(M_{i+1}))$. This means that each maintenance transaction is not allowed to commit until its preceding query has successfully begun. Therefore, a serializable schedule can be achieved if the integrity constraint $t(c(M_i)) < t(b(Q_i)) < t(c(M_{i+1}))$ is not violated. The warehouse state is incrementally maintained by a series of consecutive maintenance transactions in response to the consistent snapshots required by incoming queries.

Figure 3 illustrates the implementation of the workload scheduler. An internal queue called *ssq* is introduced for a serializable schedule of maintenance and query transactions. Each element e in *ssq* represents the status of corresponding transaction and serves as a waiting point to suspend the execution of its transaction. We also introduced the three levels of query consistency (i.e. `open`, `closed` and `complete`) defined in [6] in our work to identify the status of maintenance transaction. At any time there is always one and only one `open` element stored at the end of *ssq* to indicate an open maintenance transaction (which is M_4 in this example). Once a query (e.g. Q_4) arrives at the workload scheduler①, the workload scheduler first changes the status of the last element in *ssq* from `open`

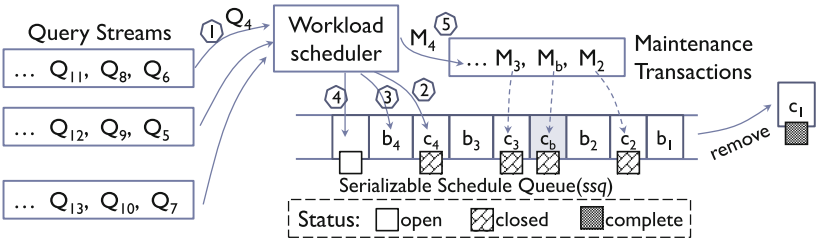

Fig. 3. Scheduling maintenance transactions and OLAP queries

to `closed`. This indicates that the scope of the input delta rows for `open` M_4 has been fixed and the commitment c_4 of M_4 should wait on the completion of this *ssq* element②. Furthermore, a new element b_4 is pushed into *ssq* which suspends the execution of Q_4 before its begin action③. Importantly, another new `open` element is created and put at the end of *ssq* to indicate the status of a subsequent maintenance transaction triggered by the following incoming query (e.g. Q_5)④. M_4 is triggered to started afterwards⑤. When M_4 is done and all the deltas have arrived at warehouse site, it marks its *ssq* element c_4 with `complete` and keeps waiting until c_4 is removed from *ssq*. Our workload scheduler always checks the status of the head element of *ssq*. Once its status is changed from `closed` to `complete`, it removes the head element and notifies corresponding suspended transaction to continue with subsequent actions. In this way, the commitment of M_4 would never precede the beginning of Q_3 which takes a consistent snapshot maintained by its preceding maintenance transactions $\{M_2, M_b, M_3\}$. Besides, Q_4 begins only after M_4 has been committed. Therefore, the constraints are satisfied and a serializable schedule is thereby achieved. It is worth to note that the workload scheduler will automatically trigger several maintenance transactions M_b to start during the idle time of data warehouses to reduce the synchronization delay for future queries. A threshold is set to indicate the maximum size of buffered deltas. Once it is exceeded, M_b is started without being driven by queries.

5 Incremental ETL Pipeline on Kettle

To achieve consistent snapshots for a sequence of queries $\{Q_i, Q_{i+1}, Q_{i+2}, ...\}$, we introduced a workload scheduler in previous section to ensure a correct order of the execution sequence of corresponding maintenance transactions $\{M_i, M_{i+1}, M_{i+2}, ...\}$. However, sequential execution of maintenance flows one at a time definitely incurs significant synchronization overhead for each query. The situation even gets worse at a very high query rate. In this section, we address the performance issue and propose an incremental ETL pipeline based on Kettle [10] for high query throughput.

Take a data maintenance flow from TPC-DS benchmark [11] as an example (see Fig. 4). An existing warehouse table called *store sales* is refreshed by

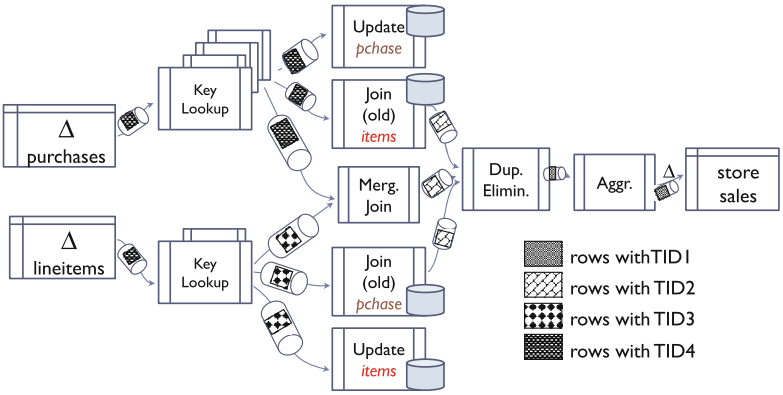

Fig. 4. Incremental ETL pipeline

an incremental ETL flow taking two delta streams *purchases* and *lineitems* as inputs. A set of key lookup steps[1] are applied on the deltas to lookup surrogate keys in warehouses. Subsequent steps (`Join (old) items`, `Merge join`, `Join (old) pchase` and `Dup. Elimin.`) form an incremental join variant to calculate deltas for previous join results. We followed a refined delta rule for natural join: $(\Delta T \bowtie S_{old}) \cup (T_{old} \bowtie \Delta S) \cup (\Delta T \bowtie \Delta S)$ introduced in [3]. By joining Δpurchases with a pre-materialized table, *items*, matching rows can be identified as deltas for *store sales*. The same holds for finding matching rows in another pre-materialized table *pchase* with new Δlineitems. After eliminating duplicates, pure deltas for incremental join can be derived plus the results of joining Δpurchases and Δlineitems. Meanwhile, in order to maintain mutual consistency in two pre-materialized tables and deliver correct results for subsequent incremental flow execution, *pchase* and *items* get updated directly with rows from respective join branches as well. After applying aggregations on the results of incremental join, final deltas for the target *store sales* table are derived.

With multiple queries $\{Q_1, Q_2, ..., Q_n\}$ occurring at the same time, *purchases* and *lineitems* streams are split into n partitions each of which is used as a specific input for the ETL flow to construct a new state s_i of *store sales* for Q_i. We refer to these partitions as *delta batches*. Before we introduce our incremental pipeline, the original implementation of Kettle is given as follows. In Kettle, each step of this incremental ETL flow is running as a single thread. As shown in Fig. 4, steps are connected with each other through in-memory data queues called *rowsets*. Rows are transferred by a provider step thread (using *put* method) through a single rowset to its consumer step thread (using *get* method). Once this incremental ETL flow gets started, all step threads start at the same time and gradually kill themselves once ones have finished with processing their batches.

Instead of killing step threads, all the steps in our incremental pipeline keep running and waiting for new delta batches. In contrast to continuous processing,

[1] ETL transformation operations are called *steps* in Kettle.

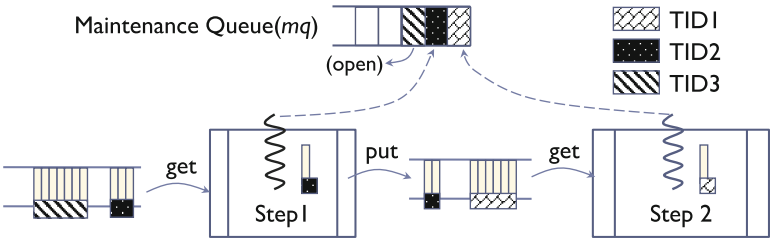

Fig. 5. Step coordination in incremental ETL pipeline

each step in our incremental pipeline must re-initialize its own state after a specific delta batch is finished. This is important to deliver consistent deltas for each query. For example, the result of sorting two delta batches together at a time could be different from sorting each batch separately. The same also holds for aggregation. Therefore, a step thread should be able to identify the scope of the rows in each delta batch, which is addressed by our incremental pipeline solution.

In Fig. 5, a two-stage pipeline is shown as an example. The incremental pipeline uses a local queue called maintenance queue *mq* to coordinate the execution of pipeline stages. The *mq* is a subset of the *ssq* described in Sect. 4 and only stores the status information of maintenance transactions. In addition, each element in *mq* stores an identifier TID_i belonging to a specific maintenance transaction M_i. Furthermore, each step thread maintains a local read point (*readPt*) and an queue index which points at a certain element in *mq*. Initially, all the incoming deltas were marked with the TID_1 of an `open` maintenance transaction. All step threads pointed at this `open` element using local indexes and waited for its status change from `open` to `closed`. With a query arrival, this element is marked as `closed` so that all the step threads are started with their the *readPt*s set to TID_1. At this time, the deltas with TID_1 are grouped to the first delta batch and become visible to step 1. After step 1 is finished with the first delta batch, it moves its queue index to the next entry in *mq* which contains TID_2. This entry is also closed by another new query arrival and step 1's *readPt* is reassigned TID_2 from the second entry. Figure 5 represents exactly a state where step 1 continues with processing the second delta batch after re-initialization while step 2 is still working on the first one. Once the second batch is done, the queue index of step 1 is moved onto a new `open` element in *mq* and the *readPt* of the step 1 is not assigned TID_3 for further processing even though there have been new deltas coming into its input delta streams.

In this way, we provide the execution of a sequence of maintenance transactions $\{M_1, M_2, ..., M_n\}$ with pipelining parallelism. Figure 4 shows a state where the example incremental pipeline is flushed by multiple delta batches at the same time to delivers consistent deltas for each Q_i. Meanwhile, the execution of each maintenance transaction starts earlier than that in sequential flow execution mode, hence reducing the synchronization overhead.

Step Synchronization in Pipeline: We address another potential inconsistency issue here caused by concurrent reads and writes to the same pre-materialized tables with different delta batches. Take the same example shown in Fig. 4. After applying four lookups on the purchase deltas, intermediate results are used to update the pre-materialized table *pchase* and meanwhile joined with another pre-materialized table *items*. The same table *pchase* and *items* are meanwhile in turn read and updated, respectively by the lineitems deltas in the other join branch. A problem might occur in this case: due to diverse processing speeds, the upper join branch could perform faster than the lower one so that the upper one has begun to update the *pchase* table using the rows belonging to the 4th TID while the same table is still being joined with the rows with the 3rd TID in the lower join branch, thus leading to anomalies and incorrect delta results for the 3rd incoming query. Without an appropriate synchronization control of these two join branches, the shared materialization could be read and written by arbitrary deltas batches at the same time.

We solve this obstacle by first identifying an explicit dependency loop across the steps in two join branches, i.e. `Update pchase` is dependent on `Join (old) pchase` and `Update items` depends on `Join (old) items`. With a known dependency loop in a pipeline, all four above mentioned steps are bundled together and share a synchronization aid called *CyclicBarrier*. Any acquisition of a new TID from *mq* by a step in this bundle would not precede the rest of associated steps. This forces all four steps to start to work on the same batch at the same time, thus leading to consistent delta output.

6 Experimental Results

We examine the performance in this section with read-/update-heavy workloads running on three kinds of configuration settings.

Test Setup: We used the TPC-DS benchmark [11] in our experiments. Our testbed is composed of a target warehouse table *store sales* (SF 10) stored in a Postgresql (version 9.4) [12] instance which was fine-tuned, set to serializable isolation level and ran on a remote machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8 GB RAM, 1 TB SATA-II disk), a maintenance flow (as depicted in Fig. 4) running locally (Intel Core i7-4600U Processor, 2×2.10 GHz, 12 GB RAM, 500 GB SATA-II disk) on Kettle (version 4.4.3) together with our workload scheduler and a set of query streams each of which issues queries towards remote *store sales* once at a time. The maintenance flow is continuously fed by deltas streams from a CDC thread running on the same node. The impact of two realistic CDC options (see Sect. 3) was out of scope and not examined.

We first defined three configuration settings as follows. **Near Real-time (NRT):** simulates a general near real-time ETL scenario where only one maintenance job was performed concurrently with query streams in a small time window. In this case, there is no synchronization of maintenance flow and queries. Any query can be immediately executed once it arrives and the consistency is not guaranteed. **PipeKettle:** uses our workload scheduler to schedule the execution

sequence of a set of maintenance transactions and their corresponding queries. The consistency is thereby ensured for each query. Furthermore, maintenance transactions are executed using our incremental ETL pipeline. **Sequential execution (SEQ):** is similar to **PipeKettle** while the maintenance transactions are executed sequentially using a flow instance once at a time.

Orthogonal to these three settings, we simulated two kinds of read-/update-heavy workloads in the following. **Read-heavy workload**: uses one update stream (SF 10) consisting of *purchases* (♯: 10 K) and *lineitems* (♯: 120 K) to refresh the target warehouse table using the maintenance flow and meanwhile issues totally 210 queries from 21 streams, each of which has different permutations of 10 distinct queries (generated from 10 TPC-DS ad-hoc query templates, e.g. q[88]). For PipeKettle and SEQ, each delta batch consists of 48 new purchases and 570 new lineitems in average. **Update-heavy workloads**: uses two update streams (♯: 20 K and 240 K) while the number of query streams is reduced to seven (totally 70 queries). Before executing a query in PipeKettle and SEQ, the number of deltas to be processed is 6-times larger than that in read-heavy workloads.

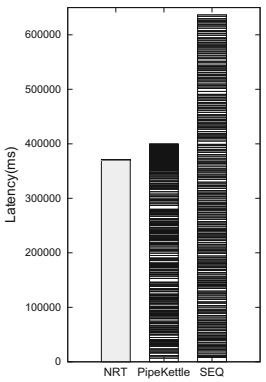

Fig. 6. Performance comparison without queries

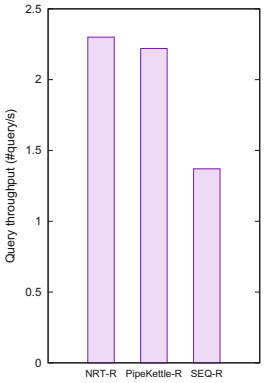

Fig. 7. Query throughput in read-heavy workload

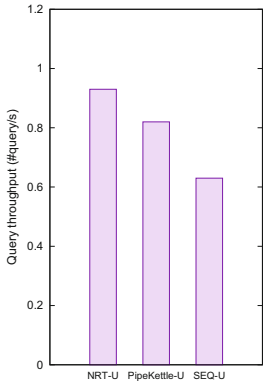

Fig. 8. Query throughput in update-heavy workload

Test Results: Fig. 6 illustrates a primary comparison among NRT, PipeKettle and SEQ in terms of flow execution latency without query interventions. As the baseline, it took 370 s for NRT to processing one update stream. The update stream was later split into 210 parts as deltas batches for PipeKettle and SEQ. It can be seen that the overall execution latency of processing 210 delta batches in PipeKettle is 399 s which is nearly close to the baseline due to pipelining parallelism. However, the same number of delta batches is processed longer in SEQ (∼650 s, which is significantly higher than the others).

Figures 7 and 8 show the query throughputs measured in three settings using both read-/update-heavy workloads. Since the delta batch size is small in read-heavy workload, the synchronization delay for answer each query is also small.

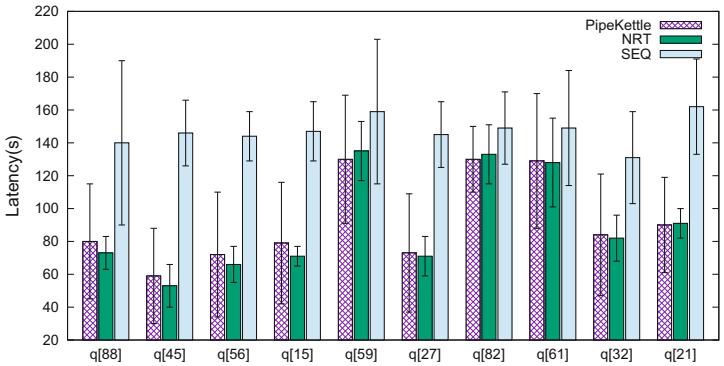

Fig. 9. Average latencies of 10 ad-hoc query types in read-heavy workload

Therefore, the query throughput achieved by PipeKettle (2.22 queries/s) is very close to the one in baseline NRT (2.30) and much higher than the sequential execution mode (1.37). We prove that our incremental pipeline is able to achieve high query throughput at a very high query rate. However, in update-heavy workload, the delta input size becomes larger and the synchronization delay grows increasingly, thus decreasing the query throughput in PipeKettle. Since our PipeKettle automatically triggered maintenance transactions to reduce the number of deltas buffered in the delta streams, the throughput (0.82) is still acceptable as compared to NRT(0.93) and SEQ (0.63).

The execution latencies of 10 distinct queries recorded in read-heavy workload is depicted in Fig. 9. Even with synchronization delay incurred by snapshot maintenance in PipeKettle, the average query latency over 10 distinct queries is approaching the baseline NRT whereas NRT does not ensure the serializability property. SEQ is still not able to cope with read-heavy workload in terms of query latency, since a query execution might be delayed by sequential execution

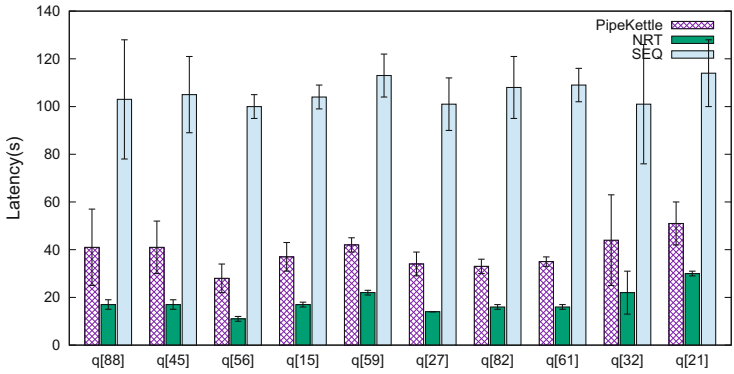

Fig. 10. Average latencies of 10 ad-hoc query types in update-heavy workload

of multiple flows. Figure 10 shows query latencies in update-heavy workload. With a larger number of deltas to process, each query has higher synchronization overhead in both PipeKettle and SEQ than that in read-heavy workload. However, the average query latency in PipeKettle still did not grow drastically as in SEQ since the workload scheduler triggered automatic maintenance transactions to reduce the size of deltas stored in input streams periodically. Therefore, for each single query, the size of deltas is always lower than our pre-defined batch threshold, thus reducing the synchronization delay.

7 Conclusion

In this work, we address the real-time snapshot maintenance in MVCC-supported data warehouse systems. Warehouse tables are refreshed by delta streams in a query-driven manner. Based on the consistency model defined in this paper, our workload scheduler is able to achieve a serializable schedule of concurrent maintenance flows and OLAP queries. We extended current Kettle implementation to an incremental ETL pipeline which can achieve high query throughput. The experimental results show that our approach achieves average performance which is very close to traditional near real-time ETL while the query consistency is still guaranteed.

References

1. Vassiliadis, P., Simitsis, A.: Near real time ETL. In: Kozielski, S., Wrembel, R. (eds.) New Trends in Data Warehousing and Data Analysis, pp. 1–31. Springer, Heidelberg (2009)
2. Karakasidis, A., Vassiliadis, P., Pitoura, E.: ETL queues for active data warehousing. In: Proceedings of the 2nd International Workshop on Information Quality in Information Systems, pp. 28–39. ACM (2005)
3. Behrend, A., Jörg, T.: Optimized incremental ETL jobs for maintaining data warehouses. In: Proceedings of the Fourteenth International Database Engineering and Applications Symposium, pp. 216–224. ACM (2010)
4. Thomsen, C., Pedersen, T.B., Lehner, W.: RiTE: Providing on-demand data for right-time data warehousing. In: ICDE, pp. 456–465 (2008)
5. Zhuge, Y., Garcia-Molina, H., Hammer, J., Widom, J.: View maintenance in a warehousing environment. ACM SIGMOD Rec. **24**(2), 316–327 (1995)
6. Golab, L., Johnson, T.: Consistency in a stream warehouse. In: CIDR, vol. 11, pp. 114–122 (2011)
7. Golab, L., Johnson, T., Shkapenyuk, V.: Scheduling updates in a real-time stream warehouse. In: ICDE, pp. 1207–1210 (2009)
8. Kemper, A., Neumann, T.: HyPer: a hybrid OLTP and OLAP main memory database system based on virtual memory snapshots. In: ICDE, pp. 195–206 (2011)
9. Kimball, R., Caserta, J.: The Data Warehouse ETL Toolkit. Wiley, Indianapolis (2004)
10. Casters, M., Bouman, R., Van Dongen, J.: Pentaho Kettle Solutions: Building Open Source ETL Solutions with Pentaho Data Integration. Wiley, Indianapolis (2010)
11. http://www.tpc.org/tpcds/
12. http://www.postgresql.org/