On-Demand Snapshot Maintenance in Data Warehouses Using Incremental ETL Pipeline

Weiping Qu(✉) and Stefan Dessloch

Heterogeneous Information Systems Group,
University of Kaiserslautern, Kaiserslautern, Germany
{qu,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 the snapshot in the data warehouse with the deltas which reside at the source side of ETL flows. Before answering a query which accesses the warehouse tables, relevant tables are first refreshed with the exact source deltas which are captured until this query arrives and haven't been synchronized with the tables yet (called on-demand maintenance). Snapshot maintenance is done by an incremental recomputation pipeline which is flushed by a set of consecutive, non-overlapping delta batches in delta streams which are split according to a sequence of incoming queries. A workload scheduler is thereby used to achieve a serializable schedule of concurrent maintenance jobs 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 a 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 [10]) 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

A. Hameurlain et al. (Eds.): TLDKS XXXII, LNCS 10420, pp. 91–112, 2017.
DOI: 10.1007/978-3-662-55608-5_5

shared data partitions. Alternatively, many data warehouses deploy the 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 execution and used during the entire query lifetime without interventions incurred by concurrent updates. For time-critical decision making, however, the snapshot taken at the warehouse side is generally stale since at the same moment, there could be deltas that haven't captured yet from the source OLTP systems or being processed by an ETL tool. In order to achieve a more refreshed snapshot, it is needed to first synchronize the source deltas with the relevant tables before taking the snapshot. Hence, a synchronization delay cannot be avoided which is incurred by an ETL flow execution.

The scope of our work is depicted in Fig. 1. We assume that a CDC process runs continuously and always pulls up-to-date changes without those maintenance anomalies addressed in [5].

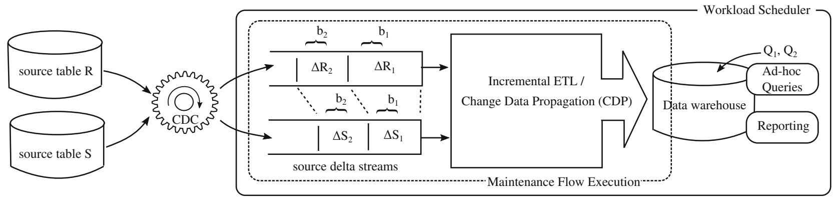


Fig. 1. Consistency scope in warehouse model

Correct and complete sets of delta tuples (Δ: insertions, updates and deletions on source tables R and S) are continuously pushed into so-called *source delta streams* in DPA. An event of a query arrival at the warehouse side triggers the system to group the current delta tuples in every source delta stream as a *delta batch* and to construct a *maintenance job* which takes the delta batches as input and perform one run of maintenance flow execution using incremental ETL techniques. The final delta batch which is produced by this maintenance flow execution is used to refresh the target warehouse tables and then a snapshot is taken for answering this query. For example, as shown in Fig. 1, the arrival of Q_1 leads to the construction of a maintenance job m_1. The input for m_1 are two delta batches b_1 with the delta tuples as ΔR_1 and ΔS_1 that are derived from the source transactions committed before the arrival time of Q_1. The query execution of Q_1 is initially suspended and later resumed when relevant tables are refreshed by the output of m_1. We call this *on-demand maintenance policy.*

With a sequence of incoming queries, a list of chained maintenance jobs are created for ETL flow to process. For efficiency and consistency, several challenges exist and are listed as follows. Firstly, sequential execution of ETL flow instances can lead to high synchronization delay at a high query rate. Parallelism needs to be exploited at certain level of the flow execution to improve performance. Furthermore, general ETL flows could contain operations or complex user-defined procedures which read and write shared resources. While running separate ETL flow/operation instances simultaneously for different maintenance jobs, inconsistency may occur due to uncontrolled access to shared resources. Finally, in our work, a warehouse snapshot S_i is considered as *consistent* for an incoming query Q_i if S_i is contiguously updated by final delta batches from preceding maintenance jobs (m_1–m_i) before the submission time of Q_i and is not interfered by fast finished succeeding jobs (e.g. m_{i+1}, which leads to non-repeatable read/phantom read anomalies). While timestamps used to extend both delta tuples 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 update sequence and OLAP queries.

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 flow executions in ETL tools and OLAP queries in warehouses. The contributions of this work are as follows:

- We introduce our on-demand maintenance policy for snapshot maintenance in data warehouses according to a computational model.
- Based on the infrastructure introduced for near real-time ETL [1], we proposed for an incremental ETL pipeline as a runtime implementation of the logical computational model using an open-source ETL tool called Pentaho Data Integration (Kettle) (shortly Kettle) [10]. The incremental ETL (job) pipeline can process a list of chained maintenance jobs simultaneously for high query throughput.
- 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.
- Furthermore, we introduce *consistency zones* in our incremental ETL pipeline to avoid potential consistency anomalies, using incremental join and slowly changing dimension maintenance as examples.
- The experimental results show that our approach achieves nearly similar performance as in near real-time ETL while the query consistency is still guaranteed.

This paper is an extended version of our previous work [16] and is structured as follows. We start by introducing terminology in our work and then describe the computational model for our on-demand maintenance policy in Sect. 2. The incremental ETL pipeline is proposed in Sect. 3 as a runtime implementation of the computational model, which addresses the performance challenge. In Sect. 4, we explain the consistency model used in our work, based on which a workload scheduler is introduced in Sect. 5 to achieve the serializability property. In Sect. 6, we address potential consistency anomalies in incremental ETL pipeline and describe the consistency zones as the solutions. We validate our approach with read-/update-heavy workloads and the experimental results are discussed in Sect. 7.

2 The Computational Model

In this section, we describe the computational model for our on-demand maintenance policy. In our work, we use a dataflow system to propagate source deltas to the data warehouse and the ETL transformation programs are interpreted as *dataflow graphs*. As shown in Fig. 2, a dataflow graph is a directed acyclic graph $G(V, E)$, in which nodes $v \in V$ represent ETL transformation operators or user-defined procedures (in triangle form), and edges $e \in E$ are *delta streams* used to transfer deltas from provider operators to consumer operators. A delta stream is an ordered, unbounded collection of delta tuples (Δ: insertions (I), deletions (D) and updates (U)) and it can be implemented as an in-memory queue, a database table or a file. There are two types of delta streams: *source delta streams* (e.g. streams for ΔR and ΔS) and *interior delta streams*. The source delta streams buffer source delta tuples that are captured by an independent CDC process and maintained in commit timestamp order in terms of source-local transactions. The interior delta stream stores the output deltas that are processed by the provider operator and at the same time, transfers them to the consumer operator. Hence, the same delta stream can be either the input or the output delta stream for two different, consecutive operators.

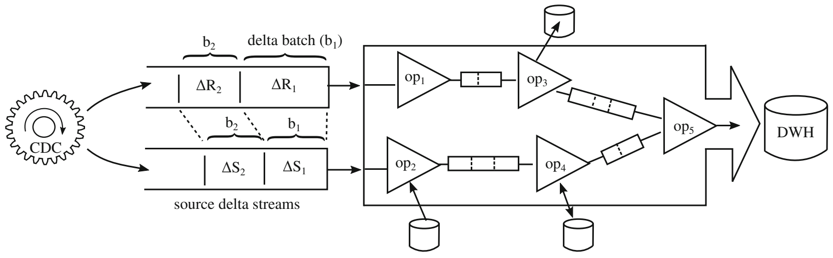


Fig. 2. Dataflow graph

Moreover, an event of a query arrival at timestamp t_i groups all source deltas with commit-time $(\Delta) < t_i$ in each source delta stream into a *delta batch* b_i and constructs a *maintenance job* m_i. Each delta batch b_i is a finite, contiguous subsequence of a delta stream and each tuple in b_i contains not only general information for incremental processing (e.g. change flag (I, D, U), change sequence number), but also the id of the maintenance job m_i. All the tuples in b_i have the same maintenance job id and should be processed together as a unit in subsequent transformation operators (e.g. op_1 and op_2). The output tuples after a delta batch processing are also assigned the same maintenance job id and are grouped into a new delta batch for downstream processing (e.g. op_3 and op_4). The maintenance job m_i is an abstraction of one maintenance flow execution where all the operators in the dataflow graph process the delta batches referring to the same job in their owning delta streams. (In the rest of the paper, we use the terms "delta batch" and "maintenance job" interchangeably to refer to the delta tuples used in one run of each transformation operator.)

With a sequence of incoming queries, the source delta streams are split to contiguous, non-overlapping delta batches and a list of chained maintenance jobs are created for the dataflow graph to process. To deliver warehouse tables with consistent deltas, the maintenance jobs needed to be processed in order in each operator. With continuous delta batches in the input delta stream, the operator execution is deployed in the following three types, depending on how the state of (external) resources is accessed.

- For operators that only write or install updates to external resources, the operator execution on each delta batch can be wrapped into a transaction. Multiple transaction instances could be instantiated for continuous incoming delta batches and executed simultaneously while these transactions have to commit in the same order as the sequence in which the maintenance jobs are created. Transaction execution protects the state from system failure, e.g. the external state would not be inconsistent in case a system crash occurs in the middle of one operator execution with partial updates. In Fig. 2, such operators can be op_3 or op_5 which continuously update the target warehouse tables. Having multiple concurrent transaction executions on incoming delta batches with a strict commit order is useful to increase the throughput.
- For operators or more complex user-defined procedures which could both read and write the same resources, transactions run serially for incoming delta batches. For example, op_4 calculates average stock price, which needs to read the stock prices installed by the transaction executions on the preceding delta batches.
- For operators that do not access any external state or probably read a private state which is rarely mutated by other applications, no transaction is needed for the operator execution. The drawback of running a transformation operator in one transaction is that the output deltas will only be visible to downstream operator when the transaction commits. To execute operators, e.g. filter or surrogate-key-lookup (op_2), no transactions are issued. The output delta batches of these operators are generated in a tuple-by-tuple fashion and can be immediately processed by subsequent operators, thus increasing the throughput of flow execution.
- A more complicated case is that multiple separate operators could access the same shared (external) resources. Thus, additional scheduling and coordination of operator executions are needed, which is detailed in Sect. 6.

3 Incremental ETL Pipeline

As introduced before, a sequence of query arrivals force our ETL maintenance flow to work on a list of chained maintenance jobs (called *maintenance job chain*), each of which brings relevant warehouse tables to the consistent state demanded by a specific query. We address the efficiency challenge of ETL maintenance flow execution in this section. We exploit pipeline parallelism and proposed an idea of *incremental ETL pipeline*.

In more detail, we define three status of a maintenance job: *pending*, *in-progress* and *finished*. When the system initially starts, a pending maintenance job is constructed and put in an empty maintenance job chain. Before any query arrives, all captured source delta tuples are tagged with the id of this job. With the event of a query arrival, the status of this pending job is changed to in-progress and all delta tuples with this job id are grouped to a delta batch as input. A new pending maintenance job is immediately constructed and appended to the end of the job chain, which is used to mark subsequent incoming source deltas with this new job id. The job ids contained in the tuples from delta batches are used to distinguish different maintenance jobs executed in the incremental ETL pipeline. The ETL pipeline is an runtime implementation of the dataflow graph where each node runs in a single, non-terminating thread (*operator thread*[1]) and each edge $e \in E$ is an in-memory pipe used to transfer data from its provider operator thread to the consumer operator thread. Each transformation operator contains a pointer which iterates through the elements in the maintenance job chain. An operator thread continuously processes tuples from incoming delta batches and only blocks if its input pipe is empty or when it points at a pending job. When the job status changes to in-progress (e.g. when a query occurs), the blocked operator thread wakes up and uses the current job id to fetch delta tuples with matching job id from its input pipe. When an operator thread finishes the current maintenance job, it re-initializes its local state (e.g. cache, local variables) and tries to fetch the next (in-progress) maintenance jobs by moving its pointer along the job chain. In this way, we construct a maintenance job pipeline where every operator thread works on its own job (even for blocking operators, e.g. sort, as well). The notion of pipelining in our case is defined at job level instead of row level. However, row-level pipelining still occurs when threads of multiple adjacent operators work on the same maintenance job.

Figure 3 illustrates a state where the ETL pipeline is flushed by four maintenance jobs (m_1–m_4). These jobs are triggered by either queries or update overload[2]. At the end of this maintenance job chain exists a pending m_6 job used to assign the id of m_6 to later captured deltas. In this example, the downstream aggregation thread has delivered target deltas of m_1 to the warehouse and blocks when it tries to work on m_2 since there is still no output from its preceding (blocking) join thread. The lookup$_2$ in the bottom join branch is still working on m_2 due to slow speed or large input size while the lookup$_1$ in the upper join branch is generating output deltas of m_3. However, the deltas with the id of m_3 in the input pipe are invisible to the join thread until it finishes

[1] As defined in Sect. 2, there are three deployment types of operator which decide whether the operator execution in this thread is wrapped in a transaction or not.

[2] We also introduce system-level maintenance jobs which are generated when the size of an input delta stream exceeds a certain threshold without any necessary query arrival. This partially hides maintenance overhead from query response time, thus shrinking the synchronization delay for answering a late query.

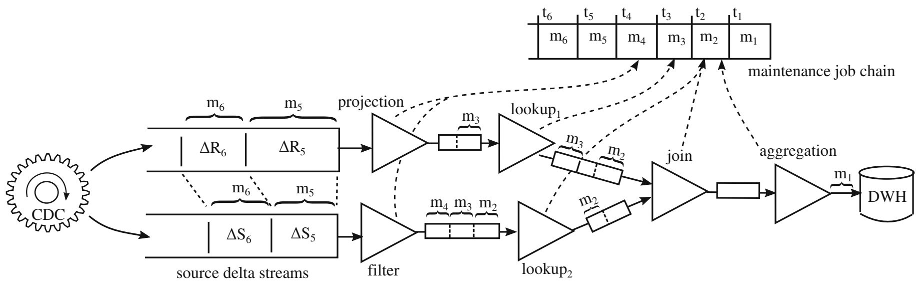


Fig. 3. Incremental ETL pipeline

m_2. Besides, a large pile-up exists in the input pipe of lookup$_2$ and more CPU cycles could be needed for it to solve transient overload. From this example, we see that our incremental ETL pipeline is able to handle continuously incoming maintenance jobs simultaneously and efficiently.

4 The Consistency Model

In this section, we introduce the notion of consistency which our work is building on. For simplicity, let us assume that an ETL flow f is given with one source table I and one target warehouse table S as sink. With an arrival of a query Q_i at point in time t_i, the maintenance job is denoted as m_i and the delta batch in the source delta stream for source table I is defined as $\Delta_{m_i}I$. After one run of maintenance flow execution on $\Delta_{m_i}I$, the final delta batch for updating the target table S is defined as follows:

$$\Delta_{m_i}S = f(\Delta_{m_i}I)$$

Given an initial state S_{old} for table S, the correct state that is demanded by the first incoming query Q_1 is derived by updating (denoted as $\uplus$) the initial state S_{old} with the final delta batch $\Delta_{m_1}S$. As defined above, $\Delta_{m_1}S$ is calculated from the source deltas $\Delta_{m_1}I$ which is captured from the source-local transactions committed before the arriving time of Q_1, i.e. t_1.

$$\begin{aligned}
S_{m_1} &\equiv S_{old} \uplus \Delta_{m_1}S \equiv S_{old} \uplus f(\Delta_{m_1}I)\\
S_{m_2} &\equiv S_{m_1} \uplus \Delta_{m_2}S \equiv S_{old} \uplus \Delta_{m_1}S \uplus \Delta_{m_2}S \equiv S_{old} \uplus f(\Delta_{m_1}I) \uplus f(\Delta_{m_2}I)\\
&....\\
S_{m_i} &\equiv S_{m_{i-1}} \uplus \Delta_{m_1}S\\
&\equiv S_{m_{i-2}} \uplus \Delta_{m_{i-1}}S \uplus \Delta_{m_i}S\\
&...\\
&\equiv S_{old} \uplus \Delta_{m_1}S \uplus \Delta_{m_2}S ... \uplus \Delta_{m_{i-1}}S \uplus \Delta_{m_i}S\\
&\equiv S_{old} \uplus f(\Delta_{m_1}I) \uplus f(\Delta_{m_2}I) ... \uplus f(\Delta_{m_{i-1}}I) \uplus f(\Delta_{m_i}I)
\end{aligned}$$

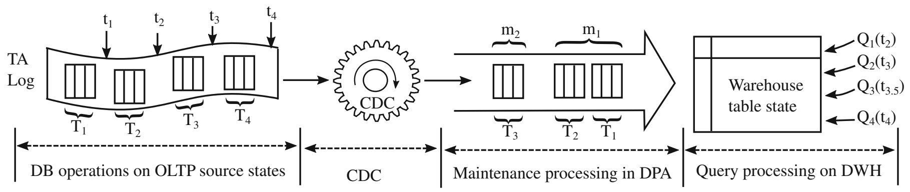


Fig. 4. Consistency model example

Therefore, we define that a snapshot of table S_{m_i} is *consistent* for the query Q_i if S_{m_i} is contiguously updated by final delta batches from preceding maintenance jobs (m_1–m_i) before the submission time of Q_i and has not received any updates from fast-finished succeeding jobs (e.g. m_{i+1}, which leads to non-repeatable read/phantom read anomalies).

An example is depicted in Fig. 4. The CDC process is continuously running and sending captured deltas from OLTP sources (e.g. transaction log) to the ETL maintenance flow which propagates updates to warehouse tables on which OLAP queries are executed. In our example, the CDC process has successfully extracted delta tuples of three committed transactions T_1, T_2 and T_3 from the transaction log files and buffered them in the DPA of the ETL maintenance flows. The first query Q_1 occurs at the warehouse side at time t_2. The execution of Q_1 is first suspended until its relevant warehouse tables are updated by maintenance flows using available captured deltas of T_1 and T_2 which are committed before t_2. The delta tuples of T_1 and T_2 are grouped together as an input delta batch with the id of the maintenance job m_1. Once m_1 is finished, Q_1 is resumed and sees an up-to-date snapshot. The execution of the second query Q_2 (at t_3) forces the warehouse table state to be upgraded with another maintenance job m_2 with only source deltas derived from 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 the final delta batch of m_1, 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 commit time of T_4. Therefore, no additional delta needs to be propagated for answering Q_3 and it shares the same snapshot with Q_2.

In our work, we assume that the CDC is always capable of delivering up-to-date changes to the 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. 4, there is a CDC delay between the recording time of T_4's delta tuples 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 the CDC has successfully delivered all required changes to the 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.

5 Workload Scheduler

As we defined the consistency notion in the previous section, the suspended execution of any incoming query resumes only if relevant tables are refreshed by corresponding final delta batch. Updating warehouse tables is normally done by the last (sink) operator in our incremental ETL pipeline and transactions are run to permanently install updates from multiple delta batches into warehouse tables. We denote the transactions running in the last sink operator thread as *sink transactions* (ST). In this section, we focus on our workload scheduler which is used to orchestrate the execution of sink transactions and OLAP queries. Integrity constraints are introduced which deliver an execution order of begin and commit actions among sink transactions and OLAP queries.

Recall that an event of a query arrival Q_i immediately triggers the creation of a new maintenance job m_i, which updates the warehouse state for Q_i. The execution of Q_i is suspended until m_i is completed in the ST_i (i.e. the i-th transaction execution of ST commits successfully with its commit action $c(ST_i)$). 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 changed by ST_i. Therefore, the first integrity constraint enforced by our workload scheduler is $t(c(ST_i)) < t(b(Q_i))$ which means that ST_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 sink transactions $\{ST_i, ST_{i+1}, ST_{i+2}, ...\}$ are run for corresponding final delta batches. Note that, once the $b(Q_i)$ successfully happens, the query Q_i does not block its successive sink transaction ST_{i+1} for consistency control since the snapshot taken for Q_i is not interfered by ST_{i+1}. Hence, $\{ST_i, ST_{i+1}, ST_{i+2}, ...\}$ can run concurrently and commit in order while each $b(Q_i)$ is aligned with the end of its corresponding $c(ST_i)$ into $\{c(ST_i), b(Q_i), c(ST_{i+1}), ...\}$. However, only with the first constraint, the serializability property is still not guaranteed since the commit action $c(ST_{i+1})$ of a simultaneous sink transaction execution might precede the begin action $b(Q_i)$ of its preceding query. For example, after ST_i is committed, the following ST_{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 $t(b(Q_i)) < t(c(ST_{i+1}))$. This means that each sink transaction is not allowed to commit until its preceding query has successfully begun. Therefore, a serializable schedule can be achieved if the integrity constraint $t(c(ST_i)) < t(b(Q_i)) < t(c(ST_{i+1}))$ is not violated. The warehouse state is incrementally maintained by a sequence of consecutive sink transactions in response to the consistent snapshots required by incoming queries.

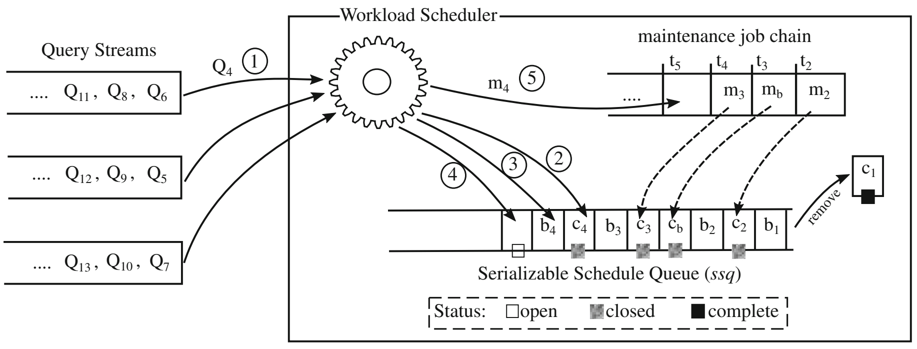


Fig. 5. Scheduling sink transactions and OLAP queries

Figure 5 illustrates the implementation of the workload scheduler. An internal queue called *ssq* is introduced for a serializable schedule of sink and query transactions. Each element *e* in *ssq* represents the status of a 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 the sink transaction. At any time there is always one and only one open element stored at the end of *ssq* to indicate an open sink transaction (which is ST_4 in this example). Once a query (e.g. Q_4) arrives at the workload scheduler ①, the workload scheduler first changes the status of the last element in *ssq* from open to closed. This indicates that the maintenance job for a pending ST_4 has been created and the commitment c_4 of ST_4 should wait on the completion of this *ssq* element ②. Furthermore, a new element b_4 is pushed into *ssq* which suspends the execution of Q_4 before its begin action ③. Importantly, another new open element is created and put at the end of *ssq* to indicate the status of a subsequent sink transaction triggered by the following incoming query (e.g. Q_5) ④. The ST_4 is triggered to be started afterwards ⑤. When ST_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 the corresponding suspended transaction to continue with subsequent actions. In this way, the commitment of ST_4 would never precede the beginning of Q_3 which takes a consistent snapshot maintained by its preceding maintenance transactions $\{ST_2, ST_b$[3], $ST_3\}$. Besides, Q_4 begins

[3] A system-level maintenance job is constructed and executed by the ST_b transaction, as certain source delta stream exceeds a pre-defined threshold.

only after ST_4 has been committed. Therefore, the constraints are satisfied and a serializable schedule is thereby achieved.

6 Operator Thread Synchronization and Coordination

In the Sect. 3, we see that the incremental ETL pipeline is capable of handling multiple maintenance jobs simultaneously. However, for those operator threads which read and write the same intermediate staging tables or warehouse dimension tables in the same pipeline, inconsistencies can still arise in the final delta batch. In this section, we first address inconsistency anomalies in two cases: incremental join and slowly changing dimensions. After that, we introduce a new concept of *consistency zone* which is used to synchronize/coordinate operator threads for consistent target deltas. In the end, we discuss the options to improve the efficiency of an incremental ETL pipeline with consistency zones.

6.1 Pipelined Incremental Join

An incremental join is a logical operator which takes the deltas (insertions, deletions and updates) on two join tables as inputs and calculates target deltas for previously derived join results. In [3], a delta rule[4] was defined for incremental joins (shown as follows). Insertions on table R are denoted as ΔR and deletions as ∇R. Given the old state of the two join tables (R_{old} and S_{old}) and corresponding insertions (ΔR and ΔS), new insertions affecting previous join results can be calculated by first identifying matching rows in the mutual join tables for the two insertion sets and further combining the newly incoming insertions found in ($\Delta R \bowtie \Delta S$). The same applies to detecting deletions.

$$\Delta(R \bowtie S) \equiv (\Delta R \bowtie S_{old}) \cup (R_{old} \bowtie \Delta S) \cup (\Delta R \bowtie \Delta S)$$

$$\nabla(R \bowtie S) \equiv (\nabla R \bowtie S_{old}) \cup (R_{old} \bowtie \nabla S) \cup (\nabla R \bowtie \nabla S)$$

For simplicity, we use the symbol Δ to denote all insertions, deletions and updates in this paper. Hence, the first rule is enough to represent incremental join with an additional join predicate `(R.action = S.action)` added to ($\Delta R \bowtie \Delta S$) where action can be insertion I, deletion D or update U.

We see that a logical incremental join operator is mapped to multiple physical operators, i.e. three join operators plus two union operators. To implement this delta rule in our incremental ETL pipeline, two tables R_{old} and S_{old} are materialized in the staging area during historical load and two extra update operators (denoted as $\uplus$) are introduced. One $\uplus$ is used to gradually maintain the staging table S_{old} using the deltas ($\Delta_{m_1}S, \Delta_{m_2}S, ...\Delta_{m_{i-1}}S$) from the executions of preceding maintenance jobs ($m_1, m_2, ..., m_{i-1}$) to bring the join table S_{old} to *consistent* state $S_{m_{i-1}}$ for $\Delta_{m_i}R$:

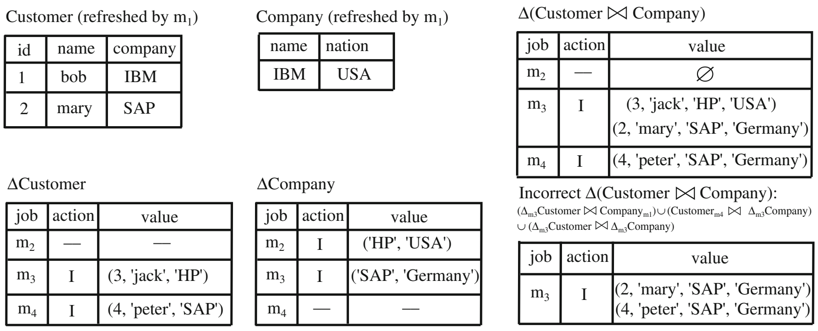
$$S_{m_{i-1}} = S_{old} \uplus \Delta_{m_1}S ... \uplus \Delta_{m_{i-1}}S$$

[4] Updates are treated as deletions followed by insertions in this rule.

Another update operator $\uplus$ performs the same on the staging table R_{old} for $\Delta_{m_i}S$. Therefore, the original delta rule is extended in the following based on the concept of our maintenance job chain.

$$\begin{aligned}\Delta_{m_i}(R \bowtie S) &\equiv (\Delta_{m_i}R \bowtie S_{m_{i-1}}) \cup (R_{m_{i-1}} \bowtie \Delta_{m_i}S) \cup (\Delta_{m_i}R \bowtie \Delta_{m_i}S)\\ &\equiv (\Delta_{m_i}R \bowtie (S_{old} \uplus \Delta_{m_{1\sim(i-1)}}S)) \cup ((R_{old} \uplus \Delta_{m_{1\sim(i-1)}}R) \bowtie \Delta_{m_i}S)\\ &\quad \cup (\Delta_{m_i}R \bowtie \Delta_{m_i}S)\end{aligned}$$

The deltas $\Delta_{m_i}(R \bowtie S)$ of job m_i are considered as *consistent* only if the update operators have completed job $m_{(i-1)}$ on two staging tables before they are accessed by the join operators. However, our ETL pipeline only ensures that the maintenance job chain is executed in sequence in each operator thread. Inconsistency can occur when directly deploying this extended delta rule in our ETL pipeline runtime. This is due to concurrent executions of join and update operators on the same staging table for different jobs.



Customer (refreshed by m_1)

id	name	company
1	bob	IBM
2	mary	SAP

Company (refreshed by m_1)

name	nation
IBM	USA

Δ(Customer ⋈ Company)

job	action	value
m_2	—	∅
m_3	I	(3, 'jack', 'HP', 'USA') (2, 'mary', 'SAP', 'Germany')
m_4	I	(4, 'peter', 'SAP', 'Germany')

ΔCustomer

job	action	value
m_2	—	—
m_3	I	(3, 'jack', 'HP')
m_4	I	(4, 'peter', 'SAP')

ΔCompany

job	action	value
m_2	I	('HP', 'USA')
m_3	I	('SAP', 'Germany')
m_4	—	—

Incorrect Δ(Customer ⋈ Company):
$(\Delta_{m3}\text{Customer} \bowtie \text{Company}_{m1}) \cup (\text{Customer}_{m4} \bowtie \Delta_{m3}\text{Company}) \cup (\Delta_{m3}\text{Customer} \bowtie \Delta_{m3}\text{Company})$

job	action	value
m_3	I	(2, 'mary', 'SAP', 'Germany') (4, 'peter', 'SAP', 'Germany')

Fig. 6. Anomaly example for pipelined incremental join

We use a simple example (see Fig. 6) to explain the potential anomaly. The two staging tables `Customer` and `Company` are depicted at the left-upper part of Fig. 6 which both have been updated by deltas from m_1. Their input delta streams are shown at left-bottom part and each of them contains a list of tuples in the form of (`job, action, value`) which is used to store insertion-/deletion-/update-delta sets (only insertions with action I are considered here) for each maintenance job. Logically, by applying our extended delta rule, consistent deltas Δ(`Customer` $\bowtie$ `Company`) would be derived which are shown at the right-upper part. For job m_3, a matching row (`'HP', 'USA'`) can be found from the company table for a new insertion (`3, 'jack', 'HP'`) on the customer table after the company table was updated by the preceding job m_2. With another successful row-matching between Δ_{m_3}Company and Customer$_{m_2}$, the final deltas are complete and correct.

However, since at runtime, each operator thread runs independently and has different execution latencies for inputs of different sizes, an inconsistent case can occur which is shown at the right-bottom part. Due to various processing costs, the join operator Δ_{m_3} Customer $\bowtie$ Company $_{m_1}$ has already started before the update operator completes m_2 on the company table, which mistakenly missed the matching row (`‘HP’, ‘USA’`) from $m2$. And the other join operator Customer $_{m_4}$ $\bowtie$ Δ_{m_3} Company accidentally reads a phantom row (`4, ‘peter’, ‘SAP’`) from the maintenance job m_4 that is accomplished by the fast update operator on the customer table. This anomaly is caused by a pipeline execution without synchronization of read-/write-threads on the same staging table.

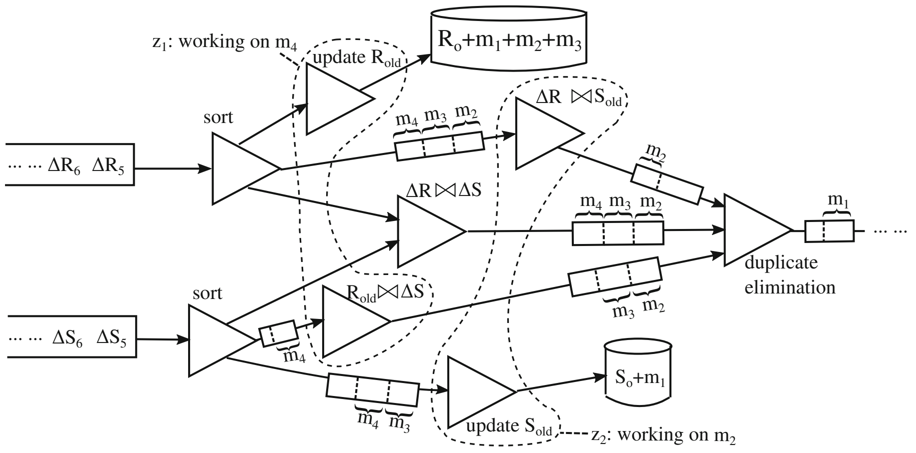


Fig. 7. Pipelined incremental join with consistency zones

To address this problem, we propose a *pipelined incremental join* for the maintenance job chain. It is supported by newly defined *consistency zones* and an extra duplicate elimination operator. Figure 7 shows the implementation of our pipelined incremental join[5]. In a consistency zone, operator thread executions are synchronized on the same maintenance job and processing of a new maintenance job is not started until all involving operator threads have completed the current one. This can be implemented by embedding a *cyclic barrier*(*cb*) (Java) object in all covered threads. Each time a new job starts in a consistency zone, this *cb* object sets a local count to the number of all involved threads. When a thread completes, it decrements the local count by one and blocks until the count becomes zero. In Fig. 7, there are two consistency zones: z_1(update-R$_{old}$, R$_{old}$ $\bowtie$ ΔS) and z_2(Δ R $\bowtie$ S$_{old}$, update-S$_{old}$), which groups together all the threads that read and write the same staging table. The processing speeds of both threads in z_1 are very similar and fast, so both of them are currently working on

[5] The two sort operators are just required for merge join and can be omitted if other join implementations are used.

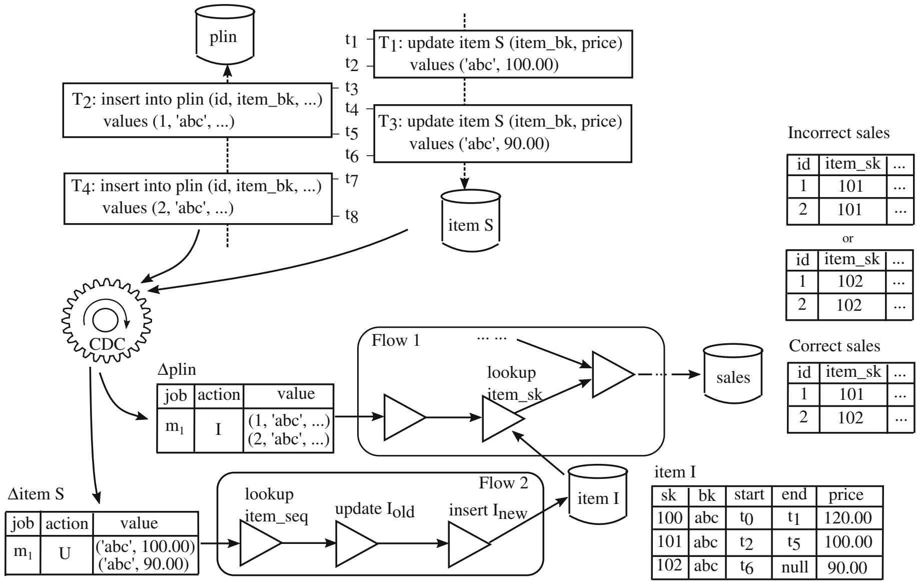
m_4 and there is no new maintenance job buffered in any of the in-memory pipes of them. However, even though the original execution latency of the join operator thread ΔR $\bowtie$ S$_{old}$ is low, it has to be synchronized with the slow operator update-S$_{old}$ on m_2 and a pile-up of maintenance jobs (m_{2-4}) exists in its input pipe. It is worth to note that a strict execution sequence of two read-/write threads is not required in a consistency zone (i.e. update-R$_{old}$ does not have to start only after R$_{old}$ $\bowtie$ ΔS completes to meet the consistency requirement R$_{m_{i-1}}$ $\bowtie$ Δ_{m_i}S). In case R$_{m_{i-1}}$ $\bowtie$ Δ_{m_i}S reads a subset of deltas from m_i (in R) due to concurrent execution of update-R$_{m_{i-1}}$ on m_i, duplicates will be deleted from the results of Δ_{m_i}R $\bowtie$ Δ_{m_i}S by the downstream duplicate elimination operator. Without a strict execution sequence in consistency zones, involved threads can be scheduled on different CPU cores for performance improvement. Furthermore, even though two consistency zones finish maintenance jobs in different paces, this duplicate elimination operator serves as a *Merger* and only reads correct input deltas for its current maintenance job, which is m_2 in the example.

6.2 Pipelined Slowly Changing Dimensions

In data warehouses, slowly changing dimension (SCD) tables need to be maintained which change over time. The physical implementation depends on the type of SCD (three SCD types are defined in [9]). For example, SCDs of type 2 are history-keeping dimensions where rows comprising the same business key represent a history of one entity while each row has a unique surrogate key in the warehouse and was valid in a certain time period (from start date to end date and the current row version has the end date null). With a change occurring in the source table of a SCD table, the most recent row version of the corresponding entity (end date is null) is updated by replacing the null value with the current date and a new row version is inserted with a new surrogate key and a time range (current date - null). In the fact table maintenance flow, the surrogate key of this current row version of an entity is looked up as a foreign key value in the fact table.

Assume that the source tables that are used to maintain fact tables and SCDs reside in different databases. A globally serializable schedule S of the source actions on these source tables needs to be replayed in ETL flows for strong consistency in data warehouses [12]. Otherwise, a consistency anomaly can occur which will be explained in the following (see Fig. 8).

At the upper-left part of Fig. 8, two source tables: `plin` and `item-S` are used as inputs for a fact table maintenance flow (`Flow 1`) and a dimension maintenance flow (`Flow 2`) to refresh warehouse tables `sales` and `item-I`, respectively. Two source-local transactions T$_1$ (start time: t$_1$ $\sim$ commit time: t$_2$) and T$_3$ (t$_4$$\sim$ t$_6$) have been executed on *item-S* to update the price attribute of an item with business key ('abc') in one source database. Two additional transactions T$_2$ (t$_3$$\sim$ t$_5$) and T$_4$ (t$_7$$\sim$ t$_8$) have been also completed in a different database where a new state of source table *plin* is affected by two insertions sharing the same business key ('abc'). Strong consistency of the warehouse state can be reached if the globally serializable schedule S: T$_1$ $\leftarrow$T$_2$ $\leftarrow$T$_3$ $\leftarrow$T$_4$ is also guaranteed


plin
T2: insert into plin (id, item_bk, ...) values (1, 'abc', ...)
T4: insert into plin (id, item_bk, ...) values (2, 'abc', ...)
t1
t2
t3
t4
t5
t6
t7
t8
T1: update item S (item_bk, price) values ('abc', 100.00)
T3: update item S (item_bk, price) values ('abc', 90.00)
item S
CDC
Δplin
job | action | value
m1 | I | (1, 'abc', ...) (2, 'abc', ...)
Δitem S
job | action | value
m1 | U | ('abc', 100.00) ('abc', 90.00)
Flow 1
lookup item_sk
sales
Flow 2
lookup item_seq
update I_old
insert I_new
item I
Incorrect sales
or
Correct sales

Incorrect sales

id	item_sk	...
1	101	...
2	101	...

or

id	item_sk	...
1	102	...
2	102	...

Correct sales

id	item_sk	...
1	101	...
2	102	...

item I

sk	bk	start	end	price
100	abc	t_0	t_1	120.00
101	abc	t_2	t_5	100.00
102	abc	t_6	null	90.00

Fig. 8. Anomaly example for ETL pipeline execution without coordination

in ETL pipeline execution. A consistent warehouse state has been shown at the bottom-right part of Fig. 8. The surrogate key (101) found for the insertion (`1, 'abc', ...`) is affected by the source-local transaction T_1 on *item-S* while the subsequent insertion (`2, 'abc', ...`) will see a different surrogate key (102) due to T_3. However, the input delta streams only reflect the local schedules S_1: $T_1 \leftarrow T_3$ on *item-S* and S_2: $T_2 \leftarrow T_4$ on *plin*. Therefore, there is no guarantee that the global schedule S will be correctly replayed since operator threads run independently without coordination. For example, at time t_9, a warehouse query occurs, which triggers an immediate execution of a maintenance job m_1 that brackets T_2 and T_4 together on *plin* and groups T_1 and T_3 together on *item-S*. Two incorrect states of the *sales* fact table have been depicted at the upper-right part of the figure. The case where *item_sk* has value 101 twice corresponds to an incorrect schedule: $T_1 \leftarrow T_2 \leftarrow T_4 \leftarrow T_3$ while another case where *item_sk* has value 102 twice corresponds to another incorrect schedule: $T_1 \leftarrow T_3 \leftarrow T_2 \leftarrow T_4$. This anomaly is caused by an uncontrolled execution sequence of three read-/write-operator threads: *item_sk-lookup* in Flow 1 and *update-I_{old}* and *insert-I_{new}* in Flow 2.

To achieve a correct globally serializable schedule S, the CDC component should take the responsibility of rebuilding S by first tracking start or commit timestamps of source-local transactions[6], mapping them to global timestamps and finally comparing them to find out a global order of actions. In addition,

[6] Execution timestamps of in-transaction statements have to be considered as well, which is omitted here.

the execution of relevant operator threads needs to be coordinated in this global order in the incremental ETL pipeline. Therefore, another type of *consistency zone* is introduced here.

Before we introduce our new consistency zone for our *pipelined SCD*, it is worth to note that the physical operator that is provided by the current ETL tool to maintain SCDs does not fulfill the requirement of the SCD (type 2) in our case. To address this, we simply implement SCD (type 2) using *update-*I_{old} followed by *insert-*I_{new}. These two operator threads need to be executed in an atomic unit so that queries and surrogate key lookups will not see an inconsistent state or fail when checking a lookup condition. Another case that matters is that the execution of Flow 1 and Flow 2 mentioned previously is not performed strictly in sequence in a disjoint manner. Instead of using flow coordination for strong consistency, all operators from the two flows (for fact tables and dimension tables) are merged into a new big flow where the atomic unit of *update-*I_{old} *insert-*I_{new} operator threads can be scheduled with the *item_sk-lookup* operator thread at a fine-grained operator level.

Our approach for pipeline coordination used in pipelined SCD is illustrated in Fig. 9. We first explain how the CDC process can help rebuild the global schedule S. Recall that a maintenance job is constructed when a query is issued or when the size of any input delta stream exceeds a threshold (see Sect. 3). We refine the maintenance job into multiple internal, fine-grained *tasks* whose construction is triggered by a commit action of a source-local transaction affecting the source table of a SCD.

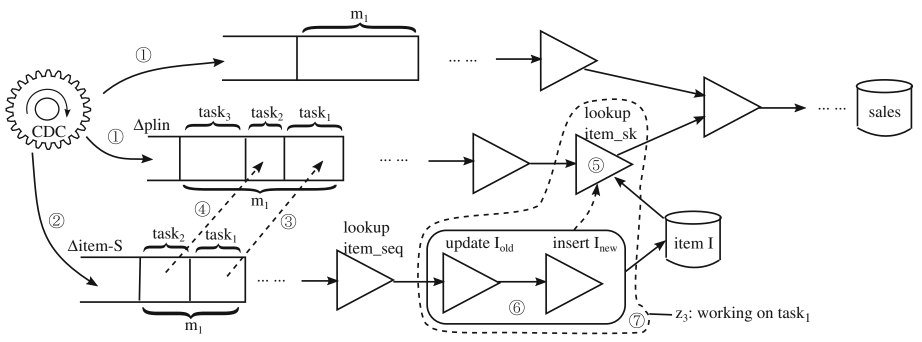


Fig. 9. Pipelined SCD with consistency zone

As shown in Fig. 9, ① the CDC continuously puts those captured source deltas into the input delta streams (one is *Δplin*) of the fact table maintenance flow. At this time, a source-local update transaction commits on *item-S*, which creates a task_1 and comprises the delta tuples derived from this update transaction ②. This immediately creates another task_1 in the input delta stream *Δplin* which contains all current available delta tuples ③. This means that all source-local, update transactions belonging to the task_1 in *Δplin* have committed before the task$_1$ of *Δitem-S*. With a commit of the second update transaction on source table *item-S*, two new task$_2$ are created in both input delta streams ④. When a query is issued at a later time, a new m_1 is constructed which contains task$_{1\sim2}$ on *Δitem-S* and task$_{1\sim3}$ on *Δplin* (delta tuples in task$_3$ commit after the task$_2$ in *Δitem-S*). During execution on m_1, a strict execution sequence between the atomic unit of *update-*I_{old} and *insert-*I_{new} and the *item_sk-lookup* is forced for each task$_i$ in m_1. The *update-*I_{old} and *insert-*I_{new} have to wait until the *item_sk-lookup* finishes task$_1$ ⑤ and the *item_sk-lookup* cannot start to process task$_2$ until the atomic unit completes task$_1$ ⑥. This strict execution sequence can be implemented by the (Java) *wait/notify* methods as a provider-consumer relationship. Furthermore, in order to guarantee the atomic execution of both *update-*I_{old} and *insert-*I_{new} at task level, (Java) *cyclic barrier* can be reused here to let *update-*I_{old} wait to start a new task until *insert-*I_{new} completes the current one ⑥. Both thread synchronization and coordination are covered in this consistency zone ⑦.

6.3 Discussion

In several research efforts on operator scheduling, efficiency improvements can be achieved by cutting a data flow into several sub-flows. In [14], one kind of sub-flow called *superboxes* are used to batch operators to reduce the scheduling overhead. And authors of [15] use another kind of sub-flow (*strata*) to exploit pipeline parallelism to some extent. In this work, the operators involved in a sub-flow are normally connected through data paths. However, as described in the previous two sections, consistency zones can have operator threads scheduled together without any connecting data path. This increases the complexity of algorithms which try to increase the pipeline efficiency as much as possible by minimizing the execution time of operator with $max(time(op_i))$. However, we will not validate the performance of the scheduling algorithms extended for consistency zones using experiments in this paper. A pipeline that was previously efficient can be slowed down dramatically when one of its operator is bound with a very slow operator in a consistency zone, which increases the $max(time(op_i))$.

The efficiency of an incremental ETL pipeline with consistency zones could be improved if the data storage supports multi-version concurrency control, where reads do not block writes and vice versa. Therefore, a fast update operator on a staging table will not be blocked by a slow join operator which reads rows using version number (possibly maintenance job id in our case). However, in another case, a fast join operator may still have to wait until the deltas with the desired version are made available by a slow update operator.

7 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) 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), an ETL pipeline (an integrated pipeline instance used to update item and store sales tables) running locally (Intel Core i7-4600U Processor, 2 × 2.10 GHz, 12 GB RAM, 500 GB SATA-II disk) on an extended version of Kettle (version 4.4.3) together with our workload scheduler and a set of query streams, each of which issues queries towards the remote store sales table 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. 4) 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 maintenance job consists of 48 new purchases and 570 new lineitems in average.

Update-heavy workloads: uses two update streams (♯: 20 K & 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.

Test Results: Figure 10 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 maintenance jobs in PipeKettle is 399 s which is nearly close to the baseline due to pipelining

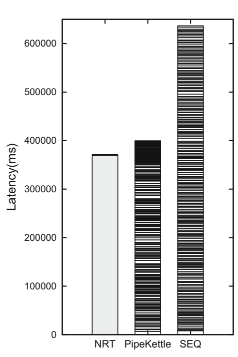


Fig. 10. Performance comparison without queries

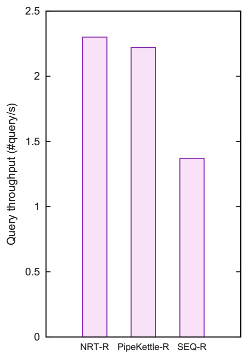


Fig. 11. Query throughput in read-heavy workload

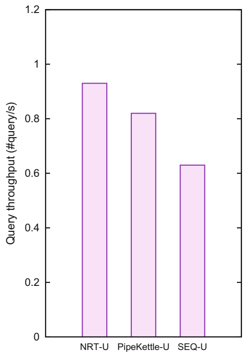


Fig. 12. Query throughput in Update-heavy workload

parallelism. However, the same number of maintenance jobs is processed longer in SEQ (~650 s, which is significantly higher than the others).

Figure 11 and 12 show the query throughputs measured in three settings using both read-/update-heavy workloads. Since the maintenance job size is small in read-heavy workload, the synchronization delay for answer each query is also small. 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

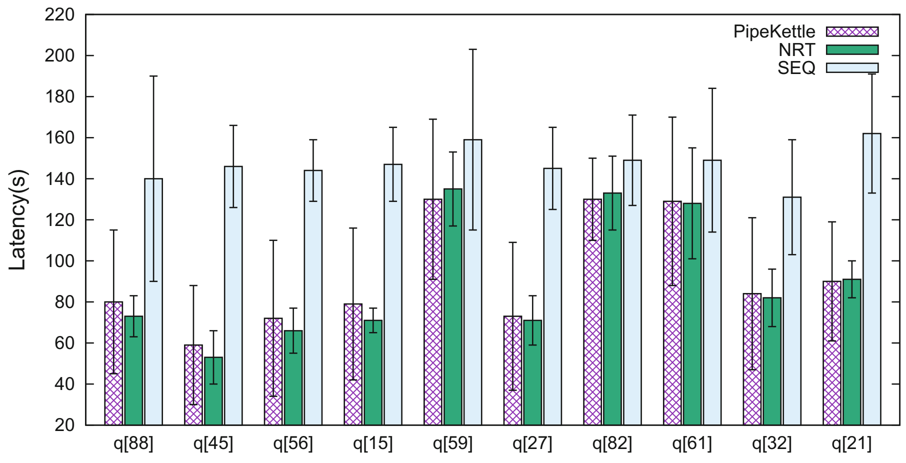


Fig. 13. Average latencies of 10 ad-hoc query types in read-heavy workload

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. 13. 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 of multiple flows.

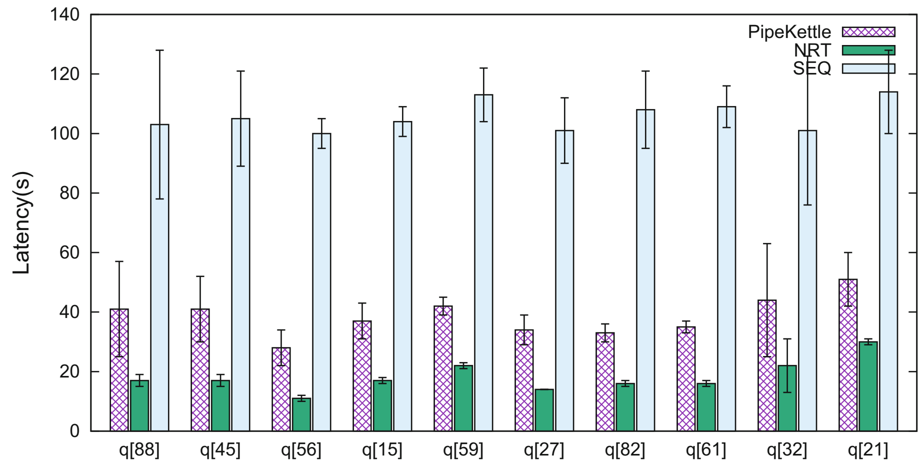


Fig. 14. Average latencies of 10 ad-hoc query types in update-heavy workload

Figure 14 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.

8 Related Work

Our incremental ETL pipeline was inspired by the work from [13] where materialized views are lazily updated by maintenance tasks when a query is issued to the database. Additional maintenance tasks are also scheduled when the system has free cycles to hide maintenance overhead partially from query response time. As we mentioned in the introduction section, ETL flows can be seen as a counterpart to the view definitions in databases where materialized view maintenance is performed in transactions to ensure the consistency property. Therefore, we addressed the potential consistency anomalies in general ETL engines which normally lack global transaction support.

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. A 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.

9 Conclusion

In this work, we addressed the on-demand snapshot maintenance policy in MVCC-supported data warehouse systems using our incremental ETL pipeline. Warehouse tables are refreshed by continuous delta batches in a query-driven manner. We discussed a logical computational model and described the incremental ETL pipeline as a runtime implementation which addresses the performance challenge. Moreover, based on the consistency model defined in this paper, we introduced the workload scheduler which is able to achieve a serializable schedule of concurrent maintenance flows and OLAP queries. We extended an open-source ETL tool (Kettle) as the platform of running our incremental ETL pipeline and also addressed potential inconsistency anomalies in the cases of incremental join and slowly changing dimension tables by proposing the consistency zone concept. 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, Boston (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 & 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&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, Hoboken (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. Zhuge, Y., Garcia-Molina, H., Wiener, J.L.: The Strobe algorithms for multi-source warehouse consistency. In: Fourth International Conference on Parallel and Distributed Information Systems, 1996, pp. 146–157. IEEE, December 1996
13. Zhou, J., Larson, P.A., Elmongui, H.G.: Lazy maintenance of materialized views. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 231–242. VLDB Endowment, September 2007
14. Carney, D., Çetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., Stonebraker, M.: Operator scheduling in a data stream manager. In: Proceedings of the 29th International Conference on Very Large Data Bases, vol. 29, pp. 838–849. VLDB Endowment, September 2003
15. Karagiannis, A., Vassiliadis, P., Simitsis, A.: Scheduling strategies for efficient ETL execution. Inf. Syst. **38**(6), 927–945 (2013)
16. Qu, W., Basavaraj, V., Shankar, S., Dessloch, S.: Real-time snapshot maintenance with incremental ETL pipelines in data warehouses. In: Madria, S., Hara, T. (eds.) DaWaK 2015. LNCS, vol. 9263, pp. 217–228. Springer, Cham (2015). doi:10.1007/978-3-319-22729-0_17