Incremental ETL Pipeline Scheduling for Near Real-Time Data Warehouses

Weiping Qu[1], Stefan Deßloch[2]

Abstract: We present our work based on an incremental ETL pipeline for on-demand data warehouse maintenance. Pipeline parallelism is exploited to concurrently execute a chain of maintenance jobs, each of which takes a batch of delta tuples extracted from source-local transactions with commit timestamps preceding the arrival time of an incoming warehouse query and calculates final deltas to bring relevant warehouse tables up-to-date. Each pipeline operator runs in a single, non-terminating thread to process one job at a time and re-initializes itself for a new one. However, to continuously perform incremental joins or maintain slowly changing dimension tables (SCD), the same staging tables or dimension tables can be concurrently accessed and updated by distinct pipeline operators which work on different jobs. Inconsistencies can arise without proper thread coordinations. In this paper, we proposed two types of consistency zones for SCD and incremental join to address this problem. Besides, we reviewed existing pipeline scheduling algorithms in our incremental ETL pipeline with consistency zones.

1 Introduction

With increasing demand for real-time analytic results on data warehouses, the frequency of refreshing data warehouse tables is increasing and the time window for executing an ETL (Extract-Transform-Load) job is shrinking (to minutes or seconds). The design of data warehouses and ETL maintenance flows is driven by not only efficiency but also data freshness considerations. For efficiency, incremental ETL techniques [BJ10] have been widely used in near real-time ETL flows and propagate deltas (insertions/deletions/updates) from source tables to target warehouse tables instead of recomputing from scratch. Incremental ETL is similar to materialized view maintenance while one of the differences is that view maintenance jobs are bracketed into internal transactions to make materialized views transactionally consistent with base tables, while ETL flows are generally executed by external tools without full transaction support. The consistency of data warehouses has been addressed in previous work [ZGMW96, TPL08, GJ11] by applying a range of techniques to ETL processes. In our work, the maintenance of warehouse tables is triggered immediately by incoming queries (referred to as *on-demand/lazy/deferred maintenance*). At the time each query arrives, it is suspended first in the system while a maintenance job is constructed and propagates only those source-local transactions that have commit timestamps preceding the query arrival time and have not yet been synchronized with warehouse tables. The query resumes execution when the warehouse tables are brought up-to-date by this maintenance job. An arrival of a sequence of queries forces our ETL flows to work on a sequence of

[1] TU Kaiserslautern, AG Heterogene Informationssysteme, qu@informatik.uni-kl.de
[2] TU Kaiserslautern, AG Heterogene Informationssysteme, dessloch@ @informatik.uni-kl.de

maintenance jobs (called *maintenance job chain*), each of which brings relevant warehouse tables to the correct state demanded by a specific query. For efficiency, we exploit pipeline parallelism and proposed an idea of *incremental ETL pipeline* in [Qu15]. Figure 1 shows an

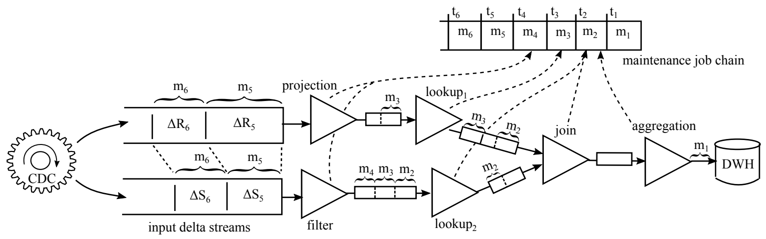


Fig. 1: Incremental ETL Pipeline

example of our incremental ETL pipeline and it consists of several components. The ETL pipeline is a data flow system and represented as a directed acyclic graph $G(V, E)$ where all nodes $v \in V$ (in triangle form) are ETL transformation operators and each edge $e \in E$ is an in-memory pipe used to transfer data from provider operator to consumer operator. The sources of the ETL pipeline are the input delta streams (one for each source table) that reside in the staging area. Each stream buffers source deltas (Δ: insertions (I), deletions (D) and updates (U)) which are captured by an independent change data capture (CDC) process and maintained in commit timestamp order. An event of a query arrival at timestamp t_i triggers the construction of a maintenance job m_{t_i} which groups the buffered source deltas with commit-time(Δ) < t_i and assigns them the id of this job. This job id is further sent to an auxiliary maintenance job chain as an id list of in-progress maintenance jobs. Each pipeline operator runs in a single, non-terminating thread and iterates through the id list. Given a job id, an operator thread blocks until input deltas with matching id occur. Once it finishes processing, it re-initializes itself and fetches the next pending job id from the job chain.

In this paper, we address pipeline scheduling for the consistency property while executing a chain of maintenance jobs using incremental ETL pipeline. We observed that data sets like staging tables or dimension tables can be read and written by different operations in the same pipeline. Take a logical incremental join (running multiple physical operators that need to access/refresh old state of the same join tables) as example. Without synchronizing or coordinating operator threads, anomalies can occur which breaks the consistency property in data warehouses. The remainder of this paper is structured as follows. We review existing scheduling algorithms from previous work mainly in ETL and stream processing domains in Section 2. In Section 3, we analyze the consistency anomalies in ETL pipelines, propose solutions called *consistency zones*, and discuss their implementations. Furthermore, we compare the experimental results of an existing scheduling algorithm (`MINIMUM COST`) with & without taking the consistency zones into account in Section 4.

2 Related Work

Previous research studies on workflow/pipeline scheduling in ETL & stream processing domains are mainly discussed here. Early work in stream processing domain addressed scheduling algorithms for execution time, throughput or memory consumption purpose. Carney et al. [Ca03] presented their scheduling algorithms in their Aurora engine which assigns operators (from continuous queries) to CPU processors (i.e. threads) at runtime based on several metrics (e.g. selectivity, processing costs, thread context switches). To reduce the scheduling and operator overheads, they group operators into *superbox*es and traverse (i.e. thread assignment) operators in a superbox in a specific order that yields minimum context switches, a lower total execution time, or maximum memory utilization. Their scheduler decides which operator is assigned a thread to process how many tuples by estimating the processing cost (i.e. tuples per time unit). This is different from our case where all operators are initially started as Java threads and scheduled originally by operating system in a round robin, time-slicing manner. Application-level scheduling can be achieved by setting priorities to threads at runtime where threads with higher priorities get longer time quantum to process than those with lower priorities.

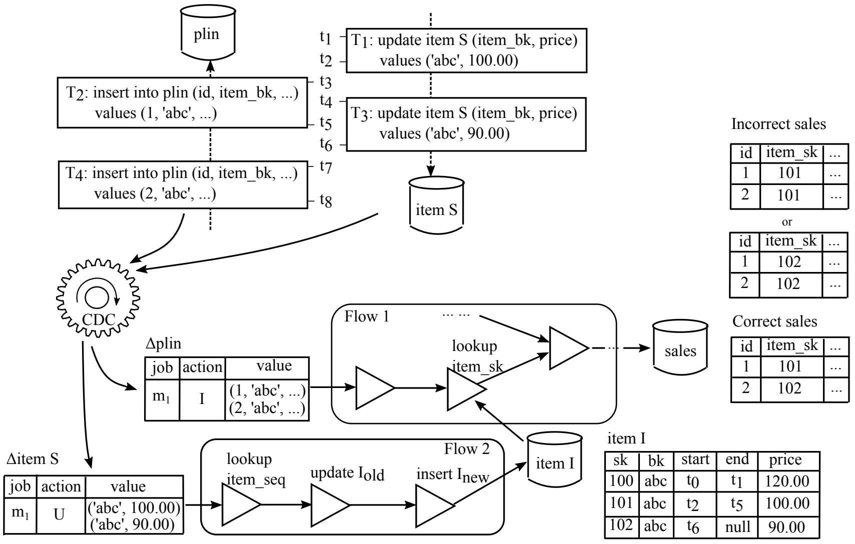
Karagiannis et al. [KVS13] also examined similar algorithms but in the (batch-oriented) ETL domain for throughput and memory consumption purposes. In their work, they exploit pipelining parallelism by dividing a large ETL workflow to connected subflows. Each subflow is a connected subgraph of original workflow and allows the data pipelining between operators. They further obtain a stratification of the subflow graph to assign mutually independent operators from subflows to subsequent layers of execution. Scheduling algorithms are applied in each subflow. For examples, their `MINIMUM COST` (MC) algorithm selects the operator with the largest volume of input data to first activate in a subflow. All work above suggested that continuous queries/ETL workflows are divided into subflows with operators connected with each other while in our work, operators in a so-called consistency zone can be separate and require to execute as an atomic group.

There exists also several work related to scheduling in data warehouses. Golab et al proposed scheduling algorithm based on the staleness metric of update jobs in streaming data warehouses [GJS12]. Resources are assigned to short/long jobs based on different policies. In [TPL08], authors introduced loading schemes to deliver trickle-updates in batch-load speed by buffering temporary incoming data either in memory on warehouse side or on client disks depending on system load. Thiele et al. [TFL09] introduced a model to schedule queries and updates at fine-grained data partition level for a balance between quality of service and quality of data in warehouses.

3 Operator Thread Coordination & Synchronization

As introduced in Section 1, the incremental ETL pipeline from our previous work is capable of handling multiple maintenance jobs simultaneously. However, potential consistency anomalies can occur in this model, which is addressed in this section for slowly changing dimensions and incremental join. Furthermore, we introduce two types of consistency zones as solutions to resolve these anomalies.

Slowly changing dimension (SCD) tables have different maintenance types. For example, SCDs of type 2 are history-keeping dimensions where multiple rows comprising the same business key can 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). However, the surrogate key of the old row version may concurrently be looked up in the fact table maintenance flow. Assume that the source tables, that are relevant to the maintenance of both 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 [ZGMW96]. Otherwise, a consistency anomaly can occur which will be explained in the following (see Figure 2). At the upper-left part of Figure 2, two source


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

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

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

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

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. 2: Anomaly Example for ETL Pipeline Execution without Coordination

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 ~ commit time: t_2) and T_3 (t_4~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~t_5) and T_4 (t_7~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 in ETL pipeline execution. A consistent warehouse state has been shown at the bottom-right part of Figure 2. 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: $T_1 \longleftarrow T_2 \longleftarrow T_4 \longleftarrow T_3$ while another case where *item_sk* has value 102 twice corresponds to another incorrect schedule: $T_1 \longleftarrow T_3 \longleftarrow T_2 \longleftarrow T_4$. 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}*, *insert-I_{new}*) in Flow 2.

The potential anomaly in **incremental join** is explained here. 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 [BJ10], a delta rule was defined for incremental joins. Let ΔR denote insertions on table R. As shown below, 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 incoming insertions found in ($\Delta R \bowtie \Delta S$). For simplicity, we use the symbol Δ here to denote all insertions I, deletions D and updates U. Hence, the rule applies to all three cases with an additional join predicate `(R.action = S.action)` added to ($\Delta R \bowtie \Delta S$), where action $\in$ {I, D, U}.

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

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 execution of preceding maintenance jobs ($m_1, m_2, ..., m_{i-1}$) to bring the join table S_{old} to the *consistent* state $S_{m_{i-1}}$ for $\Delta_{m_i}R$:

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

Another update operator $\uplus$ performs the same maintenance 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.

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

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 both staging tables before they are accessed by the join operators. However, without further precautions, our ETL pipeline only ensures that the maintenance job chain is executed in sequence in each operator thread. Inconsistencies 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.

We use a simple example (see Figure 3) to explain the potential anomaly. The two staging tables `Customer` and `Company` are depicted at the left-upper part of Figure 3 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

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 $\bowtie$ Company)

job	action	value
m_2	—	$\varnothing$
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 $\bowtie$ Company):
(Δ_{m3}Customer $\bowtie$ Company$_{m1}$) $\cup$ (Customer$_{m4}$ $\bowtie$ Δ_{m3}Company) $\cup$ (Δ_{m3}Customer $\bowtie$ Δ_{m3}Company)

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

Fig. 3: Anomaly Example for Pipelined Incremental Join

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 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 differences in 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 and has mistakenly missed the matching row (`'HP', 'USA'`) from $m2$. And the other join operator Customer$_{m_4}$$\bowtie$$\Delta_{m_3}$Company accidentally reads a *phantom* row (`4, 'peter', 'SAP'`) from the maintenance job m_4 that is produced 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.

Consistency zone is a subgraph of the original flow graph. Operator nodes in a consistency zone do not have to be interconnected via data pipes while they always affect the same *shared mutable objects* (e.g. dimension/staging tables). To change the state of shared mutable objects using a chain of maintenance jobs in a consistent manner, the completeness of a maintenance job is synchronized in a zone while the actual execution sequence of inner nodes depends of the type of the consistency zone, e.g. in a specific order or in parallel. The concept of consistency zone is similar to nested transactions.

Pipelined Slowly Changing Dimension aims at a correct globally serializable schedule S. The CDC component participates in rebuilding S by first tracking start or commit timestamps of source-local transactions[3], mapping them to global timestamps and finally

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

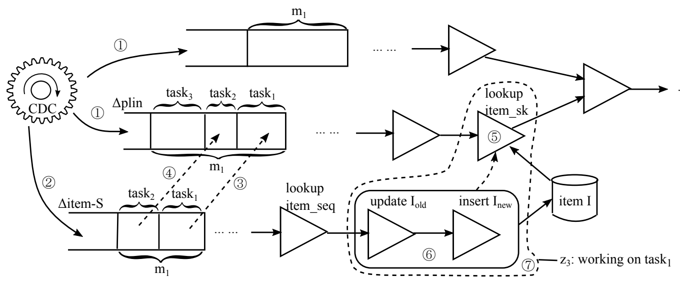


Fig. 4: Pipelined SCD with Consistency Zone

comparing them to find out a global order of actions. In addition, the execution of relevant operator threads needs to be coordinated in this global order in the incremental ETL pipeline. Therefore, a newly defined consistency zone is illustrated in Figure 4[4]. 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 Section 1). 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. As shown in Figure 4, ① the CDC continuously puts those captured source deltas into the input delta streams (one is Δ*plin*) 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 Δ*plin* which contains all current available delta tuples ③. This means that all source-local, update transactions belonging to the task$_1$ in Δ*plin* have committed before the task$_1$ of Δ*item-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 Δ*item-S* and task$_{1\sim3}$ on Δ*plin* (delta tuples in task$_3$ commit after the task$_2$ in Δ*item-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, a (Java) *cyclic barrier*[5] object can be used here to let *update-I_{old}* wait to start a new task until *insert-I_{new}*

[4] It is worth to note that the current ETL tool does not provide direct implementation of the SCD (type 2) maintenance. 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.

[5] One cyclic barrier object (*cb*) can be embedded in those threads that need to be synchronized on the completion of the same job/task. Each time a new job/task starts , 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.

completes the current one ⑥. Both thread synchronization and coordination are covered in this consistency zone ⑦.

Pipelined Incremental Join is supported by two *consistency zones* and an extra duplicate elimination operator. The consistency zone synchronizes the read-&write-threads on the same maintenance job and a new maintenance job is not started until all involving threads have completed the current one. Figure 5 shows the implementation of our pipelined incremental join. There are two consistency zones: z_1(update-R$_{old}$, R$_{old}$ $\bowtie$ ΔS) and

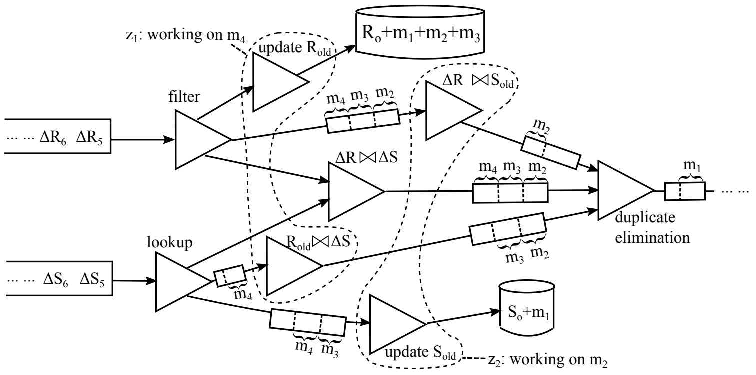


Fig. 5: Pipelined Incremental Join with Consistency Zones

z_2(ΔR⋈S$_{old}$, update-S$_{old}$), each of which maintains a (Java) *cyclic barrier* object to synchronize reads and writes on the same staging table for each job. The processing speeds of both threads in z_1 are very similar and fast, so both of them are currently working on 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⋈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 ($m_{2\sim4}$) 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⋈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, only the m_2 part of their outputs is visible to the downstream duplicate elimination operator since it currently works on m_2.

4 Consistency-Zone-Aware Pipeline Scheduling

According to the scheduling algorithm called `MINIMUM COST` (MC) [KVS13] described in Section 2, an ETL workflow is divided into subflows (each of which allows pipelining operators) and the operator having the largest volume of input data is selected to execute in each subflow. In Figure 5, a possible fragmentation results in following operator groups[6]: (filter, update-R_{old}, $\Delta R \bowtie S_{old}$), (lookup, $R_{old} \bowtie \Delta S$, update-S_{old}), ($\Delta R \bowtie S_{old}$) and (duplication elimination). However, in incremental ETL pipeline, efficiency can degrade due to synchronized execution of threads in consistency zones. The performance of a very fast pipelined subflow can drop significantly if one of its operators hooks a separate slow operator in a consistency zone outside this subflow. The side effect of consistency zones determines that they perform like blocking operations. Hence, all operators in consistency zones should be grouped together to new subflows as (update-R_{old}, $R_{old} \bowtie \Delta S$), ($\Delta R \bowtie S_{old}$, update-S_{old}), etc., which is called consistency-zone-aware MC.

Experiments: we compared the original MC and consistency-zone-aware MC here and examine the latencies of maintenance jobs in two system settings where input delta streams have a low or high input ratio, respectively (system load reaches its limit with a high input ratio). TPC-DS benchmark (www.tpc.org/tpcds) was used for experiments. The testbed comprised the fact table *store sales* (of scale factor 1), surrounding dimension tables and two staging tables materialized during historical load for pipelined incremental join. The data set is stored in a Postgresql (version 9.5) on a remote machine (2 Quad-Core Intel Xeon Processor E5335, 4×2.00 GHz, 8GB RAM). Two maintenance flows (used to maintain the *store sale* fact table and the *item* dimension table) were merged into an incremental ETL (job) pipeline (see Figure 5) that ran locally (Intel Core i7-4600U Processor, 2×2.10 GHz, 12GB RAM) in our pipeline engine which is extended from the original Pentaho Kettle (version 4.4.3, www.pentaho.com) engine. A local CDC thread[7] simulated a low input ratio (150 tuples/s) and a high input ratio (700 tuples/s), respectively. Besides, another thread continuously issued queries to the warehouse, which triggered the constructions of maintenance jobs in random time intervals. In each setting with different scheduling policies, we collected the execution time as job latency (in seconds) for 70 maintenance jobs.

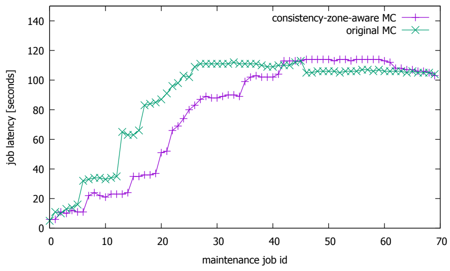


Fig. 6: input delta ratio: 150 tuples/s

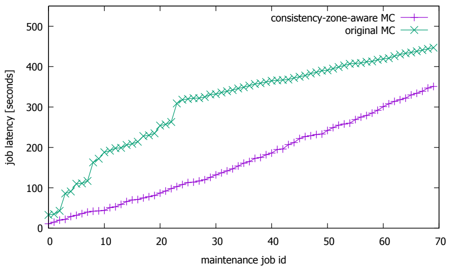


Fig. 7: input delta ratio: 700 tuples/s

Both figure 6 and 7 show that original MC performs worse than consistency-zone-aware MC, especially under high input ratio. The reason why original MC performs slow is due to the fact that the processing cost of the lookup operator was much slower, which causes starvations of downstream $R_{old} \bowtie \Delta S$ and update-S_{old}, as they are grouped in the same subflow as defined in original MC. However, the input pipes of update-R_{old} and $\Delta R \bowtie S_{old}$ grow drastically since they block due to our consistency zone features. More time quanta were assigned to them, which is not necessary and reduces the processing quantum of the

[6] blocking operations are subflows of their own.
[7] ran continuously to feed the input delta streams with source deltas to update the store sales and item table.

slow lookup operator. Hence, our consistency-zone-aware MC addresses this problem and groups the threads in consistency zones together to execute.

5 Conclusion

Based on an incremental ETL pipeline engine, we explained the potential consistency anomalies for incremental joins and slowly changing dimensions using easy-to-understand examples and proposed consistency zones with appropriate implementations. Furthermore, we moved a step towards extending previous scheduling algorithm with consistency zone features using experiments. We leave a detailed validation of further scheduling algorithms as future work.

References

[BJ10] Behrend, Andreas; Jörg, Thomas: Optimized incremental ETL jobs for maintaining data warehouses. In: Proceedings of the Fourteenth International Database Engineering & Applications Symposium. ACM, pp. 216–224, 2010.

[Ca03] Carney, Don; Çetintemel, Uğur; Rasin, Alex; Zdonik, Stan; Cherniack, Mitch; Stonebraker, Mike: Operator scheduling in a data stream manager. In: Proceedings of the 29th international conference on Very large data bases-Volume 29. VLDB Endowment, pp. 838–849, 2003.

[GJ11] Golab, Lukasz; Johnson, Theodore: Consistency in a Stream Warehouse. In: CIDR. volume 11, pp. 114–122, 2011.

[GJS12] Golab, Lukasz; Johnson, Theodore; Shkapenyuk, Vladislav: Scalable scheduling of updates in streaming data warehouses. IEEE Transactions on knowledge and data engineering, 24(6):1092–1105, 2012.

[KVS13] Karagiannis, Anastasios; Vassiliadis, Panos; Simitsis, Alkis: Scheduling strategies for efficient ETL execution. Information Systems, 38(6):927–945, 2013.

[Qu15] Qu, Weiping; Basavaraj, Vinanthi; Shankar, Sahana; Dessloch, Stefan: Real-Time Snapshot Maintenance with Incremental ETL Pipelines in Data Warehouses. In: International Conference on Big Data Analytics and Knowledge Discovery. Springer, pp. 217–228, 2015.

[TFL09] Thiele, Maik; Fischer, Ulrike; Lehner, Wolfgang: Partition-based workload scheduling in living data warehouse environments. Information Systems, 34(4):382–399, 2009.

[TPL08] Thomsen, Christian; Pedersen, Torben Bach; Lehner, Wolfgang: RiTE: Providing on-demand data for right-time data warehousing. In: 2008 IEEE 24th International Conference on Data Engineering. IEEE, pp. 456–465, 2008.

[ZGMW96] Zhuge, Yue; Garcia-Molina, Hector; Wiener, Janet L: The Strobe algorithms for multisource warehouse consistency. In: Parallel and Distributed Information Systems, 1996., Fourth International Conference on. IEEE, pp. 146–157, 1996.