Datenbank Spektrum (2014) 14:97–106
DOI 10.1007/s13222-014-0155-0

SCHWERPUNKTBEITRAG

A Real-time Materialized View Approach for Analytic Flows in Hybrid Cloud Environments

Weiping Qu · Stefan Dessloch

Received: 31 January 2014 / Accepted: 24 April 2014 / Published online: 22 May 2014

Abstract Next-generation business intelligence (BI) enables enterprises to quickly react in changing business environments. Increasingly, data integration pipelines need to be merged with query pipelines for real-time analytics from operational data. Newly emerging hybrid analytic flows have been becoming attractive which consist of a set of extract-transform-load (ETL) jobs together with analytic jobs running over multiple platforms with different functionality.

In traditional databases, materialized views are used to optimize query performance. In cross-platform, large-scale data transformation environments, similar challenges (e.g. view selection) arise when using materialized views. In this work, we propose an approach that generates materialized views in hybrid flows and maintains these views in a query-driven, incremental manner. To accelerate data integration processes, the location of a materialization point in a transformation flow varies dynamically based on metrics like source update rates and maintenance cost in terms of flow operations. Besides, by picking up the most suitable platform for accommodating views, for example, materializing and maintaining intermediate results of Hadoop jobs in relational databases, better performance has been shown.

Keywords Real-time data management · Hybrid analytic flows · Incremental view maintenance

W. Qu (✉) · S. Dessloch
Heterogeneous Information Systems Group, University of Kaiserslautern, Kaiserslautern, Germany
e-mail: qu@informatik.uni-kl.de

S. Dessloch
e-mail: dessloch@informatik.uni-kl.de

1 Introduction

Modern cloud infrastructure (e.g. Amazon EC2) aims at key features like performance, resiliency, and scalability. By leveraging NoSQL databases and public cloud infrastructure, business intelligence (BI) vendors are providing cost-effective tools in the cloud for users to gain more benefits from increasingly growing log or text files. Hadoop clusters are normally deployed to process large amounts of files. Derived information would be further combined with the results of processing operational data in traditional databases to reflect more valuable, real-time business trends and facts.

In traditional BI systems, operational data is first extracted from sources, cleansed, transformed, integrated and further loaded into a centralized database as a materialized copy for reporting or analytics. This process is referred to as a data integration pipeline and usually contains extract-transform-load (ETL) jobs. To maintain the copy in the warehouse, ETL processes are triggered to run periodically (e.g. daily) in a batch-oriented manner. In addition, another variant called incremental ETL [1] can be used, which propagates only the change data captured from the source to the target tables in a warehouse. As the size of change data is normally smaller than the original data set, incremental ETL is sometimes much more efficient than fully reloading the whole source data set. Thus, incremental ETL jobs are allowed to be scheduled in a mini-/micro-batch (i.e. hourly/in several minutes) fashion.

With the advent of next-generation operational BI, however, existing data integration pipelines cannot meet the increasing need of real-time analytics because complex ETL jobs are usually time-consuming. Dayal et al. mentioned in [2] that more light-weight data integration pipelines are expected so that a back-end integration pipeline could be

merged with a front-end query pipeline and run together as generic flows for operational BI. In [3], Simitsis et al. described their end-to-end optimization techniques for such generic flows to alleviate the impact of cumbersome ETL processes. Generic flows can be deployed to a cluster of execution engines and one logical flow operation can be executed in the most appropriate execution engine according to specific objectives, e.g. performance, freshness. These generic flows are referred to as *hybrid flows*.

On one hand, in traditional BI, offline ETL processes *push* and materialize operational source data into a dedicated warehouse where analytic queries do not have to compete with OLTP queries for resources. Therefore, query latency is low, whereas data might get stale. On the other hand, hybrid flows use data federation techniques to *pull* operational data directly from data sources on demand for online, real-time analytics. However, due to on-the-fly execution of complex ETL jobs, analytic results might be delivered with poor response times. A trade-off exists here between query latency and data freshness.

Incremental ETL could be exploited instead of fully re-executing ETL jobs in hybrid flows. A relatively small amount of change data is used as input for incremental ETL instead of the original datasets. For some flow operations, for example, filter or projection, incremental implementations outperform full reloads. However, for certain operations (e.g. join), the cost of increment variants varies on metrics (e.g. change data size) and sometimes could be worse than that of original implementation.

Therefore, we argue that just using incremental loading to maintain warehouse tables is not enough for real-time analytics. Usually, there is a *long way* (complex, time-consuming ETL jobs) from data sources through ETL processes to the warehouse. As ETL processes consist of different types of operations, incremental loading does not always perform stably. The location of the target tables (i.e. in data warehouse) has become a restriction on view maintenance at runtime using incremental loading techniques. Therefore, new thinking has emerged to relieve the materializations (in virtual/materialized integration) from their locations (data sources or the warehouse).

In this work, we propose a real-time materialized view approach in hybrid flows to optimize performance and meanwhile guarantee data freshness using on-demand incremental loading techniques. Depending on several metrics like source update rate and original/incremental operation cost from different platforms, we define a metric called *performance gain* for us to decide where the materialization point would be in the flow and which flow operators are supposed to be executed in which way, i.e. incrementally or purely. Furthermore, we explore various platform support for accommodating derived materialized views. In Sect. 2 we expose the problem that incremental recomputations do not always perform stably and the performance of certain operations changes from platform to platform. In Sect. 3 we present our main approach of real-time materialized view approach in hybrid flows. By defining the performance gain metric, we are able to meet the challenge of view selection and view deployment. The results of the experiments will be shown in Sect. 4.

2 Related Work

Hybrid Flows. Dayal et al. analyzed the requirements of new generation BI in [2] and proposed that data transformation flows should be merged into general analytic flows for online, real-time analytics. They also mentioned about using materialized views to increase performance. In [3], Simitsis et al. introduced optimization techniques in *hybrid flows* that run in a multiple-engine environment for specific objectives (e.g. performance, data freshness, fault tolerance, etc.). But they did not describe in detail how views should be selected and managed in hybrid flows. In this work, we introduce view selection and view deployment methods in hybrid flows and consider not only the performance characteristics of operations on different platforms but also the source update rates to find the *sweet spots* in hybrid flows as materialization points.

Real-time Data Warehousing. In [4], Oracle proposed their real-time data integration approach which provides a real-time copy of the transactional data in staging area for real-time reporting and maintains it incrementally. Besides, a real-time ETL solution called Morse [5] is proposed at Facebook for real-time analytics. HBase is used as underlying storage for incrementally updated table for batch processing in Hadoop/Hive. In contrast, our work treats the challenge at a fine-gained operation level and compares the operational performance with observed source update rates.

Incremental View Maintenance. In the database research community, incremental view maintenance has been extensively studied [6–8]. Furthermore, there is related work of view selection approaches concerning update cost [9, 10]. However, in ETL pipelines, certain complex transformation jobs cannot always be extended to incremental variants for efficient view maintenance at runtime. In contrast to a single database, in cross-platform environment, the main challenge we addressed in this work is to characterize the operations on multiple platforms in terms of performance and incremental implementations. In particular, choosing appropriate platforms for accommodating derived views also plays an important role in our approach.

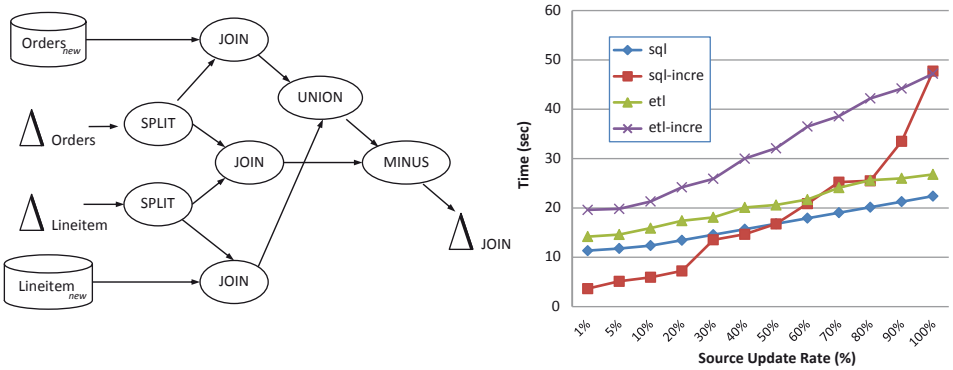


Fig. 1 Incremental join (*left*) and performance comparison on `etl` and `rdb` (*right*)

3 Incremental Recomputation in Materialized Integration

In this section, we discuss incremental recomputation techniques used in materialized views and ETL processes. We argue that incremental recomputations do not always bring low latency and the case changes from platform to platform.

Griffin et al. [11] proposed *delta rules* based on relational algebra expressions to propagate the changes/deltas captured from base tables to materialized views. For example, the delta rules of a selection $\sigma_p(S)$ and a natural join $S \bowtie T$ are $\sigma_p(\Delta(S))$ and $[S_{new} \bowtie \Delta(T)] \cup [\Delta(S) \bowtie T_{new}]$, respectively ($S, T$ denote two relations, Δ denotes change data). Given source deltas as input, new deltas are calculated at each operational level in the view query plan until the final view deltas are derived (called *incremental view maintenance*). In data warehouse environments, warehouse tables are populated by (daily) ETL processes. Similar to maintaining materialized view incrementally, Griffin's delta rules can be also applied to certain logical representations (like Operator Hub Model [12]) of ETL processes to propagate only deltas from source tables to warehouse tables.

An incremental join variant is illustrated on the left side of Fig. 1. As shown, to calculate the delta Δ_{JOIN} for an old result of table *orders* joining table *lineitem*, new insertions Δ_O on table *orders* are first joined with the new state of table *lineitem*, i.e. L_{new}, and further put together with the join result of Δ_L and O_{new} as a union. Afterwards, the rows of $\Delta_L \bowtie \Delta_O$ have to be subtracted from the union result to eliminate duplicates. We ran a small test to observe the latencies of joining *lineitem* and *orders* (scale factor 2) in a relational database `rdb` (Postgresql[1]) and an ETL tool `etl` (Pentaho Kettle[2]) on a single-node machine (2 Quad-Core Intel Xeon Processor E5335, 4 × 2.00 GHz, 8 GB RAM, 1 TB SATA-II disk) using both incremental and original implementations. During the test, we steadily increased the size of delta inputs which simulates the source update rates.

We found (shown on the right side of Fig. 1) that, in `rdb`, incremental join outperforms original join only with up to 50 % update rate (`sql` vs. `sql-incre`). With continuously increased update rates, the performance of incremental join degrades dramatically. We observed that when the delta input is small enough to fit into memory, an efficient hash join can be used with an in-memory hash table. If not, a slower sort-merge join instead will be used which contains an additional pre-sort step. The performance of two hash joins, each of which has a small join table ($\Delta < 50\,\%$), can be more efficient than one sort-merge join on two big tables. However, with the same input size, performing two sort-merge joins ($\Delta > 50\,\%$) is definitely slower than one sort-merg join. Surprisingly, in `etl` case (`etl` vs. `etl-incre`), the performance of incremental join could never compete with that of original one. Since the version of Pentaho Kettle we used does not support hash join, merge join was used which introduces additional sort stages. With limited support on join, the number of join dominates the overall performance and the performance of incremental join is always lower than the original one.

From the experimental results, we see that the performance of incremental recomputations depends on several metrics. Source update rates are important as they decide the size of input deltas which further affects the execution cost. In addition, platform support and execution capabilities play important roles as well. Furthermore, the delta rules of certain operations can also restrict efficient incremental implementations. In next section, we will describe our real-time materialized view approach for hybrid flows based on a running example.

4 Real-time Materialized Views in Hybrid Flows

Throughout this paper, we will use a running example to illustrate our real-time materialized view approach in hybrid

[1] http://www.postgresql.org.
[2] http://kettle.pentaho.com/.

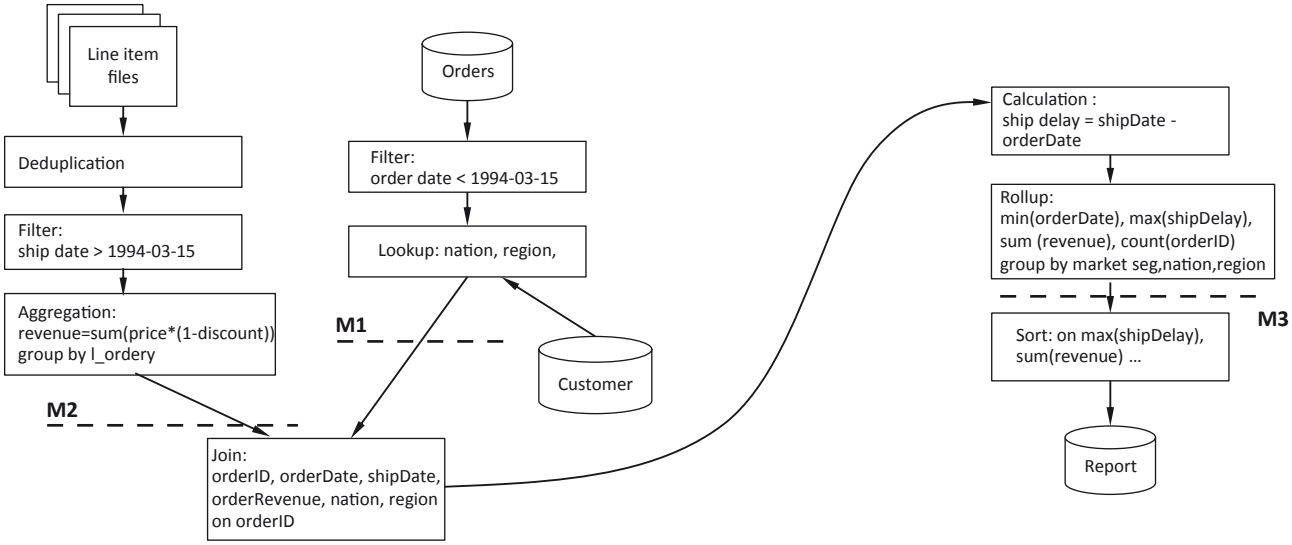


Fig. 2 An example flow checking states of incomplete orders

flows proposed in the following. Consider the sample analytic flow depicted in Fig. 2. It enables a business manager to check the state of incomplete orders in a specific branch, nation or region. Here can he prioritize the tasks with important metrics (e.g. the earliest order, the longest delay, total amount of revenues, and the number of orders) shown in the flow results and assign them to regional managers. In particular, real-time trend of buying and logistics can be captured by running analytic flow over source datasets and reactions will be immediately taken in case of emergency.

The sample flow extracts data from the `orders` table and the `line item` files and captures incomplete orders whose order date is earlier than the current date and at least one item of which has ship date later than the current date. One Lookup stage is first used to append `customers' nation/region` to the order stream. Line items are joined with orders after duplicate elimination and revenue calculation. Aggregation functions `min, max, sum, count` are then applied to the join results with group keys `branch, nation, region`. The insight is at last shown to a business manager with results sorted on important attributes (e.g. `max(shipDelay)`).

This logical flow describes the transformation and operation semantic on each stage. At flow runtime, multiple platforms can get involved. For example, large amounts of line item files with structure information can be loaded into a relational database where indexes and efficient processing are provided. They can be also quickly loaded into a distributed file system and processed by Hadoop jobs in parallel. Besides, order table and customer table may reside in different databases and an ETL tool is needed to combine information from multiple sources together. Depending on the overlap in functionality among different platforms, a single logical stage operation can be deployed to different engines. The challenge of optimizing analytic flows spanning multiple platforms (referred to as hybrid flows) has already been addressed in [3]. Optimizing approaches like data/function shipping can cut back to techniques used in distributed query processing [13]. The main idea is to ship operation to the platform which has its most efficient implementation. If the required datasets are stored on another remote platform, datasets will be shipped from remote to local only on the condition that local execution still outperforms remote one with additional data moving cost. Otherwise, this operation will be executed remotely.

To keep the business manager informed of real-time insight on incomplete orders, this hybrid flow needs to be executed repeatedly. However, even if there are relatively small changes (e.g. insertions, updates, deletions) on the source side, the old portion of source datasets still needs to get involved in the flow execution again to recompute new results. Such repeating work slows down the flow execution and meanwhile introduces large interferences and high loads to individual source systems. Furthermore, we observed that when executing flow on Hadoop platform, in many cases, the output of a chain of MapReduce(`MR`)/Hadoop jobs that process a large amount of unstructured data is well structured and significantly smaller than the original input. Such an intermediate result suits better in a relational database instead of subsequent execution in Hadoop.

Many ad-hoc hybrid analytic flows have overlap in flow operations especially those in data integration pipeline which provides consolidated data. The observations above expose improvement potential of setting up materialized views in overlapping fragments of hybrid flows. To ensure the data consistency at flow runtime, which means providing the

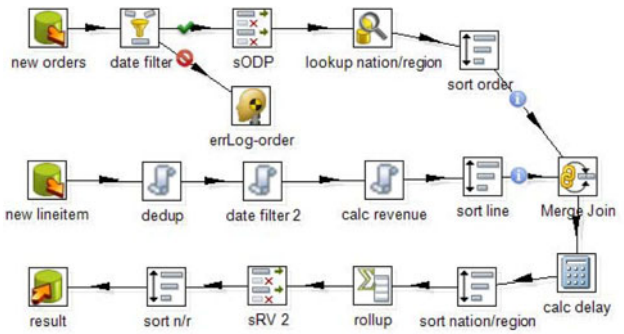


Fig. 3 Original flow

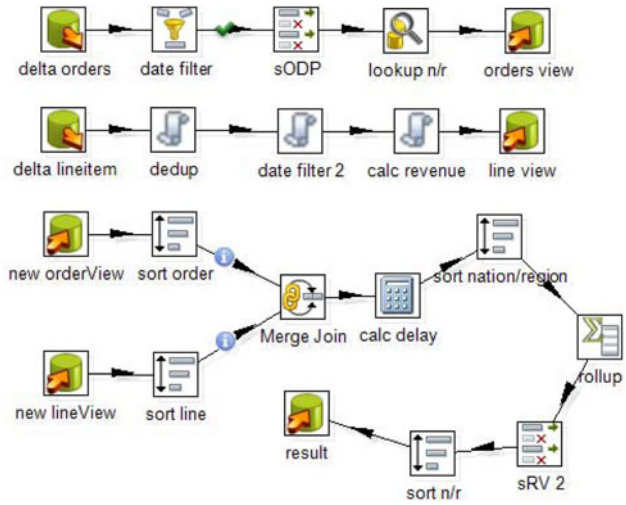


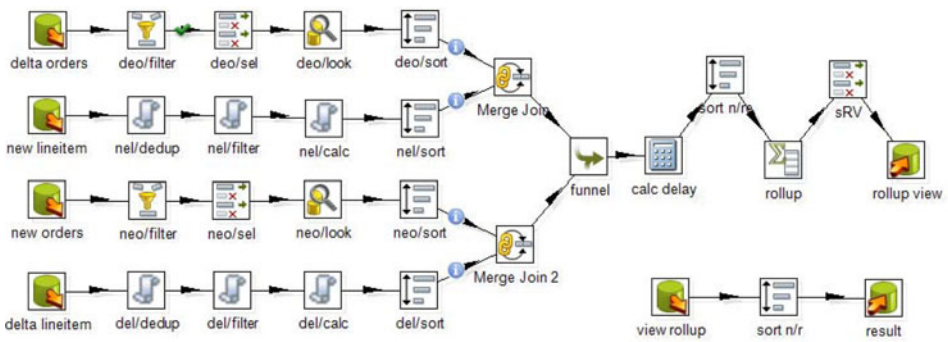
Fig. 4 Optimized flow with materialized views before join

same view of sources as executing the flow against original datasets, view maintenance has to take place on demand. This process can benefit from incremental recomputations through which only the deltas on source datasets are taken as inputs. The logical hybrid flow discussed above is now depicted by an ETL design tool and the original flow instance is shown in Fig. 3. Each logical operation is mapped to a corresponding step supported by this tool. For example, join is explicitly implemented as sort merge join in the flow for fast processing. Note that this is just a representation of the logical flow and some flow fragments are still allowed to be deployed to more efficient platforms other than ETL tool. To pick up a boundary in the flow for a materialized view, the cost of incremental implementations of flow operations will be taken into account. Recall that, as we showed in Sect. 3, incremental recomputations of specific operations (e.g. joins) sometimes perform worse than original execution and performance varies on different platforms. Assuming that the system is capable of capturing change data from sources, the flow instance shown in Fig. 4 has two materialized views set up on the boundaries (`M1, M2`) in logical flow. Intermediate results are materialized from previous execution of the two fragments and maintained with the deltas of the order table and the line item files in an incremental manner by the operations before the boundary at next runtime. Meanwhile, the subsequent join operator accesses the data from these two views for further execution using original join implementation. As we see from the experimental results in Sect. 3, the ETL tool `etl` can be used as the execution platform for this flow since the incremental join variant does not have advantage on `etl`, whereas another flow instance which has incremental join for maintaining view on the boundary `M3` (see Fig. 5) can only be deployed to the relational database `rdb` where the join performance can be guaranteed if the source update rates are lower than 50 %.

4.1 Performance Gain

With on-demand view maintenance at runtime, data freshness is guaranteed. In order to reduce the flow latency, the cost of each operator needs to be concerned into view selection step in terms of incremental and original execution. Let u_{org} denote operation u of original type, u_{inc} represent incremental type and $c(u_t)$ be latency of single operator. u_{inc} is different from u_{org} in terms of the size of input (delta vs. original) and the implementation according to delta rules mentioned in Sect. 3. Thus, the latency difference $d(u)$ between incre-

Fig. 5 Optimized flow with materialized views after join



mental and original implementations of single operator is given by the formula:

$$d(u) = c(u_{org}) - c(u_{inc}). \tag{1}$$

To simplify representing the latency of flow execution, here we only consider the flow with operation running in sequence. For specific platforms providing parallelism for subflow execution, for example, executing two fragments of a join in parallel, the cost of parallelism here is the latency of the fragment with the maximum cost. The execution latency of original flow is the sum of latency of u_{org} as follows:

$$c(F_{org}) = \sum_{i=1}^{n} c_i(u_{org}). \tag{2}$$

With our approach, an original flow is cut into two fragments by a materialized view V. The preceding fragment consists of operations of incremental type and the following fragment has the rest of operations of original type. As we let the materialized view to be maintained at runtime, extra overhead $c_{online}(V)$ is introduced to load calculated deltas into the old view and read data from the new view for subsequent operations. Therefore, the latency of view-optimized flow is shown in the following

$$c(F_V) = \sum_{i=1}^{m} c_i(u_{inc}) + c_{online}(V) + \sum_{i=m+1}^{n} c_i(u_{org}). \tag{3}$$

To have a feeling about how many benefits a materialized view on a certain boundary in a hybrid flow can bring, we define the performance gain $g(V)$ for each created view V. In following formula, $c_{offline}(V)$ represents the cost of building up a materialized view offline, e.g. the storage waste on the platform accommodating this view. A weight σ is used here to make $c_{offline}(V)$ comparable to flow latency. As one-off view generation could benefit multiple future hybrid flow, we put another variable λ as a weight to enhance view effect on performance gain. Based on the Formulae 1, 2, 3 defined above, we give the formula of the performance gain of the materialized view as follows:

$$\begin{aligned} g(V) &= \frac{\lambda \times (c(F_{org}) - c(F_V)) - \sigma \times c_{offline}(V)}{c(F_{org})} \\ &= \frac{\lambda \times \left(\sum_{i=1}^{m} d_i(u) - c_{online}(V)\right) - \sigma \times c_{offline}(V)}{c(F_{org})} \end{aligned} \tag{4}$$

Performance gain represents the rate of the improvement achieved from view-optimized flows as compared to the performance of original flow. As the latency of view-optimized flow increases, the performance gain decreases. In case F_V contains operations of incremental type which dramatically degrades the whole performance and leads to $c(F_{org}) \leq c(F_V)$, no performance gain exists. Furthermore, even if $c(F_{org}) - c(F_V)$ is positive, significant cost in creating views $c_{offline}(V)$ can also affect $g(V)$.

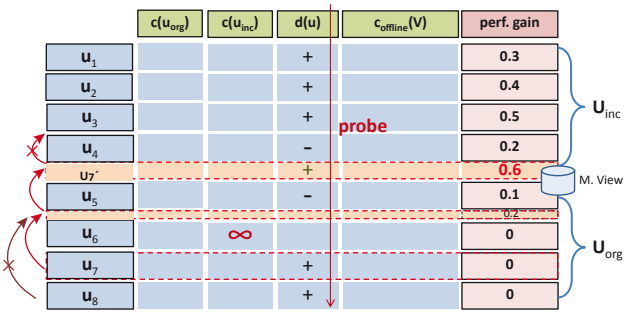


Fig. 6 Selecting view in a single-platform flow

4.2 View Selection and Deployment

In this section, we will discuss how to implement our real-time materialized view approach in a hybrid flow. Two main challenges, i.e. view selection and view deployment, are involved. View selection is referred to as the process of finding out a position to insert a materialized view in the flow. This will be discussed only in a single-platform environment. In a cross-platform environment, hybrid flow spans multiple platforms with different functionality. One problem left is which platform should be chosen to accommodate the view for the highest execution efficiency. This will be considered as view deployment problem.

View Selection is described in the following. Figure 6 shows a selection table with a list of flow operations which are sorted according to their positions in the input flow.

To initialize this table, during previous flow execution, the latency of each operation with original implementation has been recorded as $c(u_{org})$. Meanwhile, the size of its intermediate result is stored into $c_{offline}(V_i)$ as extra storage cost. Based on the change data capture mechanism, update rate on each source is allowed to be captured, which implies that $c(u_{inc})$ can be simulated with the size of given deltas. According to the formulae defined in previous section, the latency difference $d(u)$ can be derived and further used to calculate the performance gain. In this example, symbols $+/-$ are simplified values for $d(u)$ instead of real values. They indicate whether the difference is positive or negative and whether incremental variant outperforms original one or not. At last, the performance gain $g(V_i)$ for each flow operation u_i is calculated based on a simulated, view-optimized flow F_{V_i} where a materialized view V_i is supposed to be inserted right after operation u_i. All preceding operations (including u_i) before V_i would run incrementally and subsequent operations after u_i would be executed as u_{org}. Thus, $g(V_i)$ is the comparison

result between the latency of this flow ($c(F_{V_i})$) and that of original flow ($c(F_{org})$).

After initial phase, we consider the potential of optimizing performance with flow transition techniques. In [14], Simitisis et al. introduced a set of logical transitions (e.g. swap, factorize, distribute, merge, split, etc.) that can be applied to ETL workflow optimization under certain condition. In this example, we only use the `SWAP` transition to interchange the sequence of adjacent operations in the flow. Starting from the first operation row in the selection table, we probe the values in the latency difference column $d(u)$ until we reach an operation row which has its $c(u_{inc})$ value as ∞ (u_6 in this example). This means that this operation is not able to support incremental recomputation, for example, sort operation and some user-defined functions. In the probe phase, u_6 is the first occurring operation which cannot be executed incrementally, which indicates that materialized view can only be set in front of it for executable incremental view maintenance. We record u_6 and probe further.

For the rest of the operations, we try to push the ones that have positive $d(u)$ in front of u_6 using multiple swap transitions since it is likely that certain operations (e.g. filter) could increase the performance gain. Two candidates u_7, u_8 have been found and u_8 failed as the movement of u_8 cannot lead to an identical state of original flow. Until this step, we have a sequence of operations (u_{1-5}, u_7) as candidates of view boundary. Operation u_3 has the maximal performance gain `0.5` estimated and u_7 has only `0.2` since there are two operations u_4, u_5 sitting in front of it. Their incremental performance is lower than that of original implementations thus degrades u_7's gain. Therefore, we try to swap them with u_7 again and now u_7 achieves higher rank (due to the same state reason, u_7 cannot be pushed in front of u_4). The $g(V)$ of u_7 turns to `0.6` as the position of u_7 in the flow affect can affect the performance gain. After recomputing the performance gain, u_7 becomes the winner as materialization boundary in the flow, even if there is u_4 in front of it. This means u_7 brings significant benefit with its incremental variant, which offsets the negative impact from u_4. Using this method, we are able to find out an appropriate boundary for materialized views in a single-platform flow.

View Deployment Recall that the execution of hybrid flow spans multiple platforms with overlap in processing functionality. Logical flow operations can be deployed to different platforms for the highest efficiency. Introducing materialized views in hybrid flows and maintaining them on the fly can achieve additional improvement. For example, it may be effective for a Hadoop cluster to produce analytic results on terabytes of unstructured files as compared to a relational database. However, by caching small size of analytic results in this relational database, it would be much more efficient for it to maintain these views with smaller update files.

By extending the view selection table in a multiple-platform environment, the cost of original and incremental operations will be compared across platforms. Those platforms that support incremental recomputations better are preferred for on-demand view maintenance. However, one problem occurs that which platform should be used for accommodating generated views.

Suppose that a materialized view is supposed to be inserted between two consecutive operations u_i and u_{i+1} (u_i run incrementally and u_{i+1} run with original impl.). If both operations are deployed to the same platform P_x, a simple way is to materialize the intermediate results directly in this platform. If u_i run on P_x and u_{i+1} on P_y, the situation becomes whether to maintain views on P_x and move views onto P_y or move deltas onto P_y and maintain the views there. In order to answer this question, we extend the formula of *placement rate* defined in [3] as follows:

$$p(V) = \frac{\sigma \times c^{P_x}_{offline}(V) + \lambda \times \left(c^{i \to i+1}_{view} + c\left(u^{P_y}_{i+1}\right)\right)}{\sigma \times c^{P_y}_{offline}(V) + \lambda \times \left(c^{i \to i+1}_{delta} + c\left(u^{P_y}_{i+1}\right)\right)}. \quad (5)$$

$c^{P}_{offline}(V)$ denotes the cost of initial loading of materialized view in platform P. $c^{i \to i+1}_{view/delta}$ indicates whether to transfer the new views or the deltas onto platform P_y where u_{i+1} takes place. As we can see, if both platforms have no problem of initializing this view, it should be placed onto the platform P_y since the latency of transferring deltas is definitely lower. In case there is no more free space for view V on platform P_y, this view may be stored in another platform P_z where $c^{i \to i+1}_{delta} + c\left(u^{P_z}_{i+1}\right)$ is lower than $c^{i \to i+1}_{view} + c\left(u^{P_y}_{i+1}\right)$.

5 Experimental Results

Test Setup To validate our approach, we measured performance of three flow variants in our running example described in Sect. 4. In this example exist three flows: *f1* has flow operations running with their original implementations over source datasets; *f2* contains two materialized views (`M1, M2`) derived from previous execution results for the flow fragments before join. The inputs are source change data captured in idle time and used to maintain two views at runtime; *f3*: has a materialized view (`M3`) set after join operation. The incremental join variant runs on the fly to calculate the deltas for the view.

Three data stores are involved in the experiment, namely `orders, customer` and `lineitem files` using TPC-H benchmark. Experimental platforms comprises a relational database (`rdb`), an open source ETL tool (`etl`) and a M/R Hadoop engine (`mr`). The experiment consists of

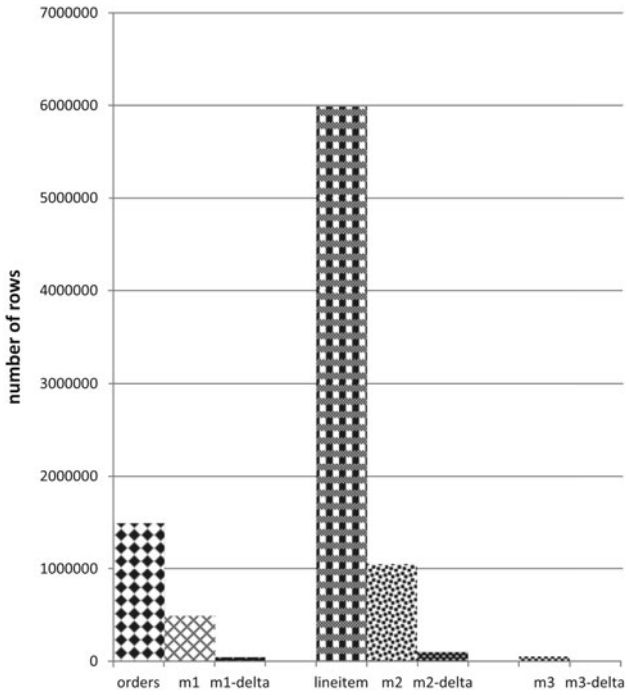


Fig. 7 Data size comparison

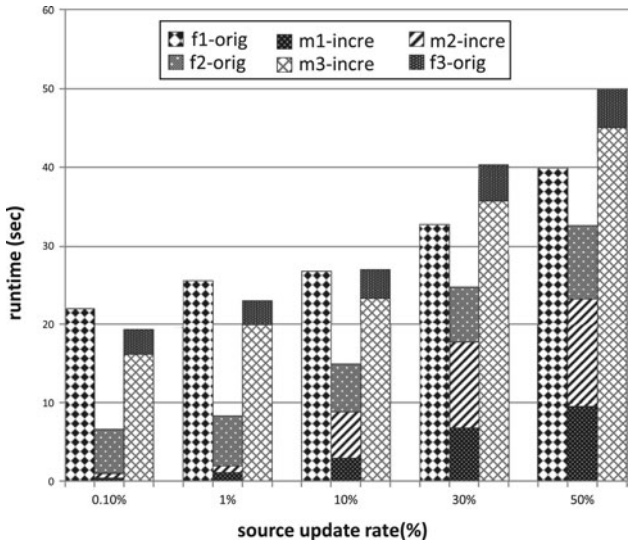


Fig. 8 Running flows (1, 2, 3) in `rdb`

two parts. In the first part, flow *f1, f2* and *f3* are tested on two single platforms `rdb` and `etl`, respectively. The goal is to prove that our view selection method can be used to find the best materialization points in transformation flows on a single platform. The second part takes flow *f2* as example to show the benefits of view deployment method. The flow fragment for processing `lineitem files` is either deployed to `mr` or first materialized and further maintained by `rdb` at runtime.

We ran our experiments in `rdb` and `etl` both running on a single-node machine (2 Quad-Core Intel Xeon Processor E5335, 4 × 2.00 GHz, 8 GB RAM, 1 TB SATA-II disk). Hadoop runs on a 6-node cluster (2 Quad-Core Intel Xeon Processor X3440, 4 × 2.53 GHz, 4 GB RAM, 1 TB SATA-II disk, Gigabit Ethernet).

View Selection on Single Platform We ran flow *f1, f2* and *f3* in `rdb` and `etl`, respectively with input datasets of scale factor 1 (i.e. 1G). For *f2* and *f3*, the update rates of `orders` and `lineitems` are 0.1, 1, 10, 30 and 50 %. Figure 7 compares the number of rows among input sources (`orders, lineitems`), derived views (`M1, 2, 3`) and the deltas with 10 % source update rate. The source table `orders` has 1.4M rows and `lineitem files` have 5.7M rows which are accessed directly by flow *f1*. As compared to *f1*, the number of the rows in the input (`M1 + M2 + deltas`) for flow *f2* is 1.6M and flow *f3* reads the smallest view (`M3 + deltas`, 60 K rows).

In Fig. 8, there are 5 groups each of which represents the results of running 3 flows in `rdb` with different source update rate. In each group, the result of flow *f2* consists of three portions. The underlying two values represent the latency of maintaining and reading `M1,2` at runtime using incremental variants, respectively. The value on top is the runtime of running subsequent operations with original implementations after reading data out of `M1,2`. Flow *f3* has only one view to maintain, thus consists of two values: maintenance time and execution time.

The results show that with increasing update rates, in `rdb`, flow *f2* always outperforms *f1*. The incremental variants of the operations *group, filter, lookup* that maintain `M1,2` in *f1* have lower latency than their original implementations, thus show high performance gain. Deltas can be calculated through each operator without accessing source data. In particular, as common *group, filter* operations have smaller output size than their input size, the size of derived materialized views is much smaller. The impact of online delta-loading and view-reading is reduced. In contrast to *f2*, flow *f3* shows worse performance and has higher latency than *f1* when the source update rate is higher than 10 %. Even if *f3* has the smallest overhead of view-reading, the impact of incremental join dominates the runtime since large-size of source data needs to be accessed for calculating deltas.

As shown in Fig. 9, the results change in `etl`. The performance gain of views in *f2* can only be guaranteed with up to 30 % update rate. Since `etl` does not has its own storage system, derived views are stored in `rdb`. The cost of moving data through network highly reduces the performance gain.

In this part of experiments, we show that the performance varies from platform to platform. It depends not only on the operations supported by specific platforms but also on the

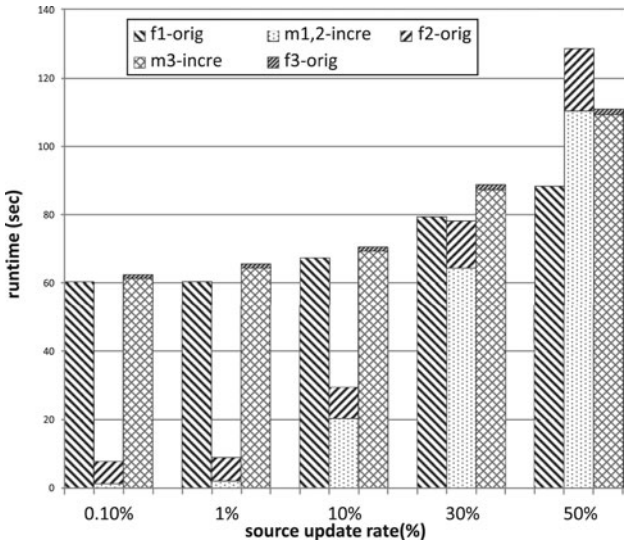


Fig. 9 Running flows (1, 2, 3) in `etl`

source update rate. Using our view selection method, the performance gain of materialized views can be calculated in a cross-platform environment with different source update rates. Thereby, the best materialization points can be suggested in the flows.

View Deployment in Hybrid Flows The second part validates the view deployment method in a hybrid flow. Data stores have scale factor 10 (i.e. 10G) in this part. Two variants of flow *f2* are used here. The first flow *f2-1* has `lineitem files` that are stored in the Hadoop distributed file system (HDFS) of `mr` and processed by Hadoop jobs before join operation. The ETL tool `etl` tool joins the results from `rdb` and `mr` and performs the subsequent operations. In the second variant *f2-2*, a materialized view `V` is created from the intermediate results of Hadoop jobs and cached in `rdb`. At runtime, view `V` is maintained by first loading update data of the `lineitem files` into `rdb` and running the incremental variants of *dedup, filter,* and *group* to calculate deltas for `V`.

The results are depicted in Fig. 10. The runtime of *f2-1* is composed of the time of executing Hadoop jobs over `lineitems` and the time of running subsequent operations in `etl`. For *f2-2*, the time is captured in load, maintenance and execution phases. With 0.1 % update rate, the load phase and maintenace phase of *f2-2* have lower latency as the delta size is small. Therefore, overall performance is significantly better than *f2-1*. As the delta size raises up to 10 %, the performance of load operations in `rdb` degrades drastrically and the overall performance is worse than that of *f2-1*.

The results give the suggestions that the overall performance can benefit from creating materialized views for Hadoop jobs in `rdb` only if the `lineitems` contain update

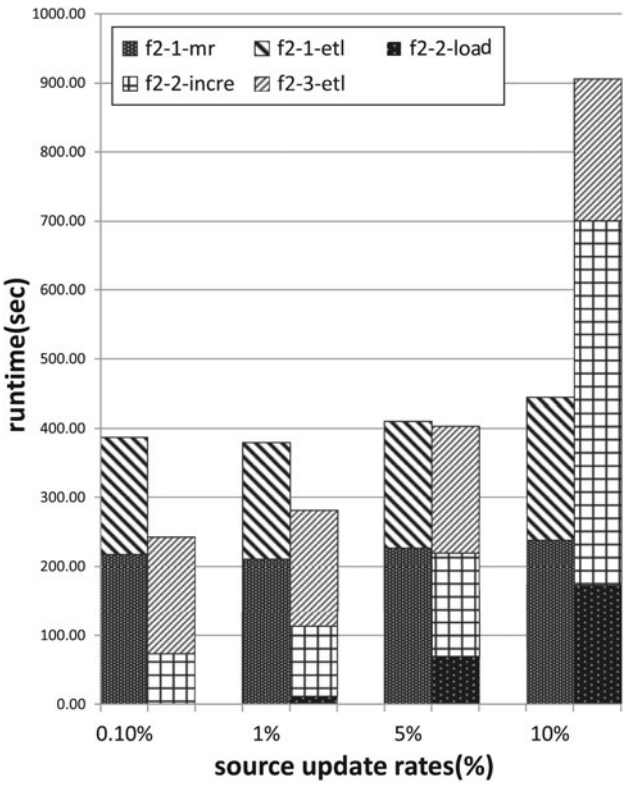


Fig. 10 Running flows 2 as hybrid flow

rates lower than 5 %. This suits better in the case where *f2-2* runs continuously. Otherwise, `mr` performs better if the delta size exceeds 5 %.

6 Conclusion

In this work, we proposed a real-time materialized view approach for data flows which span multiple platforms like Hadoop, ETL and relational databases. Materialized views are maintained by incremental recomputation techniques at runtime. As the performance of certain incremental implementations (e.g. join) varies on the operations supported by specific platforms and the source update rates, we first defined a metric called performance gain to identify the benefit of a materialized view in any possible point of a flow. Based on the performance gain, the flow operations that can achieve efficient incremental execution performance are pushed near sources using view selection method. The operations that have worse performance or are not even supported in an incremental way are pushed after materialized views and executed with original implementations. This solution works better in a heterogeneous environment where ETL flows extract data from sources with different update rates. With varying source update rates, the suggested materialization point moves in the flow dynamically. Furthermore, by using view deployment method, derived views can be deployed to ap-

propriate platforms for efficient maintenance at runtime. This method exploits the platforms with different support on incremental recomputations in hybrid flows.

The results showed that our real-time materialized view approach is able to give suggestions of using materialized views in a sequential flow execution environment where data consistency is guaranteed. In a concurrent flow execution environment, multiple flows can access the views at the same time thus skew data can read by certain flows. View update anomaly and query anomaly can take place in such case. This would be our future work.

References

1. Joerg T, Dessloch S (2008) Towards generating ETL processes for incremental loading. In: IDEAS '08 Proceedings of the 2008 international symposium on Database engineering & applications 101–110
2. Dayal U, Castellanos M, Simitsis A, Wilkinson K (2009) Data integration flows for business intelligence. EDBT '09 Proceedings of the 12th international conference on extending database technology: advances in database technology, 1–11
3. Simitsis A, Wilkinson K, Castellanos M, Dayal U (2012) Optimizing analytic data flows for multiple execution engines. SIGMOD '12 Proceedings of the 2012 ACM SIGMOD international conference on management of data, 829–840
4. Oracle white paper (2012) Best practices for real-time data warehousing
5. http://strataconf.com/stratany2013/public/schedule/detail/30630. Accessed: 1 Nov. 2013
6. Blakeley JA, Larson PA, Tompa FW (1986) Efficiently updating materialized views. ACM SIGMOD Record 15:61–71
7. Gupta A, Mumick IS (1995) Maintenance of materialized views: problems, techniques, and applications. IEEE Data Eng Bull 18:3–18
8. Zhuge Y, Garcia-Molina H, Hammer J, Widom J (1995) View maintenance in a warehousing environment. ACM SIGMOD Record 24:316–217
9. Gupta H (1997) Selection of views to materialize in a data warehouse. ICDT '97 Proceedings of the 6th international conference on database theory, 98–112
10. Hanson EN (1987) A performance analysis of view materialization strategies. SIGMOD '87 Proceedings of the 1987 ACM SIGMOD international conference on Management of data 440–453
11. Griffin T, Libkin L, Trickey H (1997) An improved algorithm for the incremental recomputation of active relational expressions. IEEE Trans Knowl Data Eng 9:508–511
12. Dessloch S, Hernandez MA, Wisnesky R, Radwan A, Zhou J (2008) Orchid: integrating schema mapping and ETL. ICDE '08 Proceedings of the 2008 IEEE 24th international conference on data engineering, 1307–1316
13. Kossmann D (2000) The state of the art in distributed query processing. ACM Comput Surv 32:422–469
14. Simitsis A, Vassiliadis P, Timos S (2005) Optimizing ETL processes in data warehouses. ICDE '05 Proceedings of the 21st international conference on data engineering, 564–575
15. Behrend A, Joerg T (2010) Optimized incremental ETL jobs for maintaining data warehouses. IDEAS '10 Proceedings of the 14th international database engineering & applications symposium, 216–224
16. Gupta A, Jagadish HV, Mumick IS (1996) Data integration using self-maintainable views. EDBT '96 Proceedings of the 5th international conference on extending database technology: advances in database technology, 140–144