Grosbeak: A Data Warehouse Supporting Resource-Aware Incremental Computing

Zuozhi Wang[†], Kai Zeng[‡], Botong Huang[‡], Wei Chen[‡], Xiaozong Cui[‡], Bo Wang[‡], Ji Liu[‡], Liya Fan[‡], Dachuan Qu[‡], Zhenyu Hou[‡], Tao Guan[‡], Chen Li[†], Jingren Zhou[‡]

[†]{zuozhiw, chenli}@ics.uci.edu, [‡]{zengkai.zk, botong.huang, wickeychen.cw, xiaozong.cxz, yanyu.wb, niki.lj, liya.fly, dachuan.qdc, zhenyuhou.hzy, tony.guan, jingren.zhou}@alibaba-inc.com

[†]UC Irvine, [‡]Alibaba Group

ABSTRACT

As the primary approach to deriving decision-support insights, automated recurring routine analytic jobs account for a major part of cluster resource usages in modern enterprise data warehouses. These recurring routine jobs usually have stringent schedule and deadline determined by external business logic, and thus cause dreadful resource skew and severe resource over-provision in the cluster. In this paper, we present Grosbeak, a novel data warehouse that supports resource-aware incremental computing to process recurring routine jobs, smooths the resource skew, and optimizes the resource usage. Unlike batch processing in traditional data warehouses, Grosbeak leverages the fact that data is continuously ingested. It breaks an analysis job into small batches that incrementally process the progressively available data, and schedules these small-batch jobs intelligently when the cluster has free resources. In this demonstration, we showcase Grosbeak using real-world analysis pipelines. Users can interact with the data warehouse by registering recurring queries and observing the incremental scheduling behavior and smoothed resource usage pattern.

ACM Reference Format:

Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, Jingren Zhou. 2020. Grosbeak: A Data Warehouse Supporting Resource-Aware Incremental Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384708

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384708

1 INTRODUCTION

Cloud-scale data warehouses are one of the keystone systems of a modern enterprise's big data infrastructure. They serve a mixed workload of recurring routine analytic jobs and ad-hoc analytic queries. As a primary approach to deriving decision-support insights, recurring routine analytic jobs account for a majority of cluster resource usages. For instance, from our observation of a data warehouse at Alibaba, up to 65% of daily workload is recurring analytic jobs. These jobs have a stringent schedule and deadline determined by various business logic. They also have similar schedules, thus causing dreadful "rush hour" schedule patterns in the cluster. Figure 1 shows a cluster resource usage pattern from a real traditional data warehouse environment in Alibaba. A majority of daily analysis queries starts after 2AM every day when the previous day's data has been fully collected. The peek usage lasts until 6AM, which is the deadline required by the business logic. During the day, there are only a few occasional spikes due to ad-hoc queries. This resource skew leads to a over-provision of resources to guarantee the peak resource usage, but leaves the cluster under utilized off traffic hours, causing excessive waste of resources.

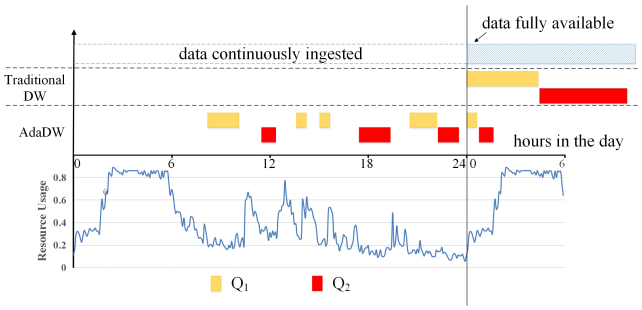

Figure 1: Timeline of running a pipeline of two queries in a traditional warehouse vs in Grosbeak

To solve this problem, we have developed Grosbeak, a data warehouse that can automatically optimize the resource usage patterns of the underlying cluster. Specifically, Grosbeak focuses on optimizing the resource-usage pattern of recurring routine analytic jobs. Traditional data warehouses predominately handle automated routine analysis using batch query processing, and schedule them strictly at the time when all the input data is available. In contrast, Grosbeak leverages the fact that the analyzed data is continuously collected and ingested, and breaks an analysis job into small batches that incrementally process the progressively available data. These incremental jobs can be scheduled as part of the data becomes available, and can be adapted according to the real-time resource usage in the cluster.

To illustrate how recurring routine analytical jobs are executed in Grosbeak, let us consider an analytic pipeline where a query Q_1 computes on ingested data, whose output is in turn used by another query Q_2. Figure 1 illustrates how the pipeline is executed in a traditional data warehouse and in Grosbeak on a timeline. Traditionally, as the data is constantly ingested, the system cannot compute Q_1 and Q_2, even though the cluster has available capacities. Differently, Grosbeak can start computing Q_1 when part of Q_1's input data is available, and the cluster has available capacities (at 10am, 12pm, 18pm etc. of a day). Each execution can resume from the work left by the previous one, and the final execution at 12am of day2 finishes much faster. Furthermore, the early outputs of Q_1 in turn enable Q_2 to execute early.

It is worth noting that dynamic scheduling in Grosbeak is very different from previous incremental computation paradigms, such as stream computing [2], incremental view maintenance [3] (IVM), etc. First, Grosbeak does not have a fixed triggering period. The scheduler can delay scheduling incremental computation if the cluster is overloaded by foreground ad-hoc workloads. In contrast, traditional IVM usually has to trigger incremental computation immediately to make sure its view is always up-to-date. Second, in between the incremental computation in Grosbeak, the recurring routine jobs do not occupy any cluster resources, and thus have much lower resource consumption. It is very different from paradigms such as stream computing, which has to keep the jobs online in order to process continuously arriving data. Third, the incremental execution in Grosbeak can be fully automated. Users do not need to manually convert their batch SQL queries into a streaming version. In addition, the data warehouse can search an optimal incremental plan from many incremental computing techniques, such as higher-order IVM etc. [1, 4] in a cost-based manner.

In this demonstration, we showcase Grosbeak using real-world analysis pipelines. Users can interact with the data warehouse by registering recurring queries, specifying hints for scheduling incremental processing, and observing the incremental scheduling behavior and the smoothed resource usage pattern in Grosbeak.

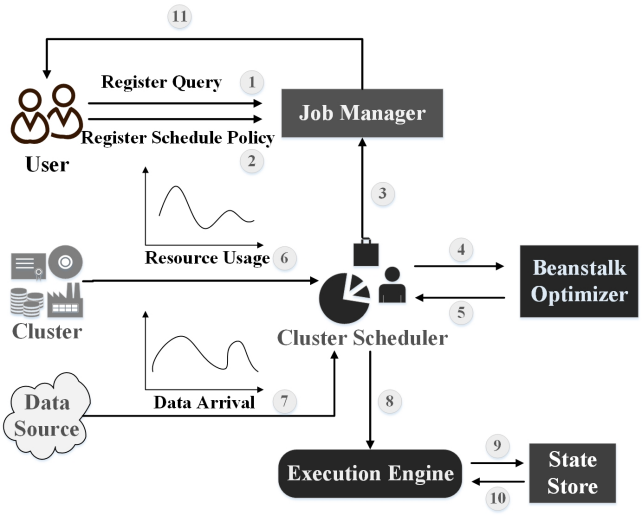

Figure 2: System architecture and job lifecycle

2 GROSBEAK SYSTEM OVERVIEW

Figure 2 shows the architecture of Grosbeak and the lifecycle of a job processed by the system. A user first registers a recurrent query on the *Job Manager* (step 1), including the query dependency in a pipeline and the deadline to produce the final result. The user can also specify a scheduling policy, such as triggering an execution when the cumulative input data passes a threshold. The user can also let Grosbeak automatically decide execution schedules. The *Job Manager* manages the whole query lifecycle and hosts the application UI. The query information is then passed to the *Cluster Scheduler* (step 3). The *Cluster Scheduler* generates a tentative schedule and invokes the *Progressive-Plan Optimizer* named Beanstalk (step 4), which uses cost-based optimization to generate a progressive plan (step 5). The plan consists of a sequence of physical execution plans to be executed at the scheduled times. The *Cluster Scheduler* starts monitoring the cluster resource usage (step 6) and data arrival for this query (step 7) in order to trigger the execution based on the cluster resource usage and user-provided scheduling policy. When an execution is triggered, the *Cluster Scheduler* submits the physical plan to the *Execution Engine* (step 8), which loads the previous intermediate states from the Grosbeak *State Store* (step 9), performs the incremental computation, and saves the new states (step 10). The *State Store* is a layered storage module that can intelligently choose a storage medium to keep the data, ranging from in-memory cache, SSD, to hard disks. This execution process (steps 8 - 10) repeats during the lifecycle of the job. The process ends when all the data has

arrived, the last job finishes execution, and the final result is delivered back to the user (step 11).

When a job is initially submitted, the cluster scheduler proposes a tentative schedule $\{t0, t1, \ldots, t*\}$ and the optimizer generates corresponding progressive execution plans $\{P0, P1, \ldots, P*\}$. This scheduling decision is based on historical information and future predictions. During the lifecycle of the job, the cluster environment might change, causing the tentative schedule to be less ideal. Grosbeak supports dynamic re-planning to handle such situations. For example, if the cluster scheduler notices that the resource usage is high at an originally scheduled time, it will delay the execution and wait for the resource utilization to drop. The user could also manually set a new scheduling policy. When the *Cluster Scheduler* decides a new execution schedule, it repeats steps 4 and 5 to invoke the optimizer to perform re-optimization. The progressive scheduler produces a new plan by jointly considering the old partially executed plan, existing intermediate states, new schedule, and updated statistics. Specifically, users can specify the desired trigger time points, e.g., (12AM, 18PM, and 22PM), or a trigger condition, e.g., when the amount of accumulated data is beyond a threshold. Such hints can let users fine-tune the scheduling decisions based on external knowledge.

3 QUERY COMPILATION

Grosbeak uses a cost-based optimizer called "Beantalk" [5] that can jointly consider (1) various incremental view maintenance techniques, (2) the statistics of available data, (3) the current cluster resources, and (4) the intermediate states from the previous runs and the schedules of the incremental jobs in the future. In this section, we present Grosbeak's compilation technique through an example, illustrate the plan space of various ways of incremental processing, and demonstrate the cost trade-off. We propose a general framework based on a model called "time-varying relation (TVR) model", in which we treat changing data as a relation varying with time. By explicitly modeling snapshots and delta views of a TVR, we can unify various incremental computation models. We elaborate the complexity of progressive query planning using an example analytic pipeline in Fig. 3(a), which computes the gross revenue of all orders by consolidating the sales orders with the returned ones. Suppose we are given a schedule S with two times t_1 and t_2, where t_2 is the specified execution time of both queries in the workflow, and t_1 is an early execution time. The data records visible at t_1 and t_2 in *sales* and *returns* are those in Fig. 3(b).

Classic incremental view maintenance. This approach treats *sales_status* and *summary* as views, and uses incremental computation to keep the views always up to date with respect to the data seen so far. At time t_1, it computes the left outer join using the snapshot at t_1 of *sales* and *returns* to generate a view *sales_status*, then applies aggregation on the join result to generate a view *summary*. At time t_2, instead of recomputing from scratch, it uses the delta input to incrementally compute the result. It first computes the delta of the left outer join by using the snapshot at t_1 and delta(t_1-t_2) of *sales* and *returns*, then performs aggregation on join's delta outputs. The delta outputs are merged with previous snapshots to update the two views. The delta input to *sales* at t_2 includes tuples $\{o_5, o_6, o_7\}$. Fig. 3(c) depicts *sales_status*'s delta outputs at t_1 and t_2, respectively, where # = +/−1 denotes insertion or deletion, respectively. Note that a *returns* record (e.g., r_2 at t_2) can arrive much later than its corresponding order (e.g., the shaded o_2 at t_1). Therefore, an *order* record may be output early as it cannot join with a *returns* record at t_1, but retracted later at t_2 when the *returns* record arrives, such as tuple o_2 of *sales_status* at t_2 in Fig. 3(c).

Non-retractable incremental computation. The approach avoids retractions in incremental computation. Specifically, in the outer join of *sales_status*, tuples in *sales* that do not join with tuples from *returns* at t_1 (o_2, o_3, and o_4) may join in the future, and thus will be held back. At time t_1, it only computes the inner join on snapshot(t_1) of *sales* and *returns* and applies the aggregation. The computation of left anti join is skipped because it produces outputs that could possibly be retracted. At time t_2, it first computes the delta of *sales* $\bowtie$ *returns* and incrementally computes the aggregation. Because t_2 is the last execution time and all data has arrived, $\gamma(sales \bowtie^{la} returns)$ can be computed, then merged with the inner join aggregation result to produce the final output.

The optimal progressive plan is data dependent, and should be determined in a cost-based way. In this example, the classic IVM approach computes 9 tuples (5 tuples in the outer join and 4 tuples in the aggregate) at t_1, and 10 tuples at t_2. Suppose the cost per unit at t_1 is 0.2 (due to fewer queries), and the cost per unit at t_2 is 1. Then its total cost is $9 \times 0.2 + 10 \times 1 = 11.8$. The non-retractable incremental computing approach computes 6 tuples at t_1, and 11 tuples at t_2, with a total cost of $6 \times 0.2 + 11 \times 1 = 12.2$. On the contrary, if the retraction is often, say, with one more *order* tuple o_4 at t_2, then the stream computing approach will become more efficient, as it costs 12.2 versus the cost 13.8 of the first approach. The reason is that retraction wastes earlier computation and causes more re-computation. Notice that the performance difference of the two approaches can be arbitrarily large.

4 DEMONSTRATION SCENARIO

This demonstration will allow the audience to interact with Grosbeak using a variety of data sets and queries, such as those from the TPC-DS benchmark. We will also provide data

```
sales_status =
  SELECT sales.o_id, category,
         price, cost
  FROM sales LEFT OUTER JOIN returns
    ON sales.o_id = returns.o_id
summary =
  SELECT category,
    SUM(COALESCE(-cost, price)) AS gross
  FROM sales_status
  GROUP BY category
```

(a)

sales

o_id	cat	price	
o_1	c_1	100	t_1
o_2	c_2	150	t_1
o_3	c_1	120	t_1
o_4	c_1	170	t_1
o_5	c_2	300	t_2
o_6	c_1	150	t_2
o_7	c_2	220	t_2

returns

o_id	cost	
o_1	10	t_1
o_2	20	t_2
o_6	15	t_2

(b)

sale_status at t_1

o_id	cat	price	cost
o_1	c_1	100	10
o_2	c_2	150	null
o_3	c_1	120	null
o_4	c_1	170	null

sale_status at t_2

o_id	cat	price	cost	#
o_2	c_2	150	null	−1
o_2	c_2	150	20	+1
o_5	c_2	300	null	+1
o_6	c_1	150	15	+1
o_7	c_2	220	null	+1

(c)

sale_status at t_1

o_id	cat	price	cost
o_1	c_1	100	10

sale_status at t_2

o_id	cat	price	cost	#
o_2	c_2	150	20	+1
o_3	c_1	120	null	+1
o_4	c_1	170	null	+1
o_5	c_2	300	null	+1
o_6	c_1	150	15	+1
o_7	c_2	220	null	+1

(d)

Figure 3: (a) A simple analytic pipeline; (b) data-arrival patterns of *sales* and *returns*; (c) results of *sales_status* produced by IVM at t_1 and t_2; and (d) results of *sales_status* produced by non-retractable computing at t_1 and t_2.

sets and query workloads that closely resemble the business analytics jobs in Alibaba.

Generating reports for an online retailing business. We demonstrate a real use case of Grosbeak in an online retailing business, which involves processing of orders, payment, and logistics data to generate daily reports. Such reports are crucial to supporting business intelligence and decision making. Data is continuously ingested into the warehouse by synchronizing with the online OLTP database change log every 15 minutes. One example application is to generate shipping status of all orders placed in a day. The query uses a left outer join of orders with logistics on the order_id attribute. Left outer join is used because only orders that are shipped out on the same day as they are placed can successfully join. An orders record is inserted immediately after an order is placed, but a logistics record is only created after a tracking number is available. In this case, Grosbeak can intelligently choose the non-retractable incremental computation approach described in Section 3 to avoid a lot of retractions. Another interesting characteristic of this workload is that most of the data arrival happens during the day, which does not coincide with the peak hours (2AM - 6AM) of the cluster. Grosbeak is able to choose an execution schedule without accumulating too much data.

Monitoring status on application dashboard. Grosbeak provides an application UI that helps users understand the progressive plans and monitor the status of the job lifecycle. Figure 4 shows a query dependency graph consisting of four queries. The user can see each query with two or three scheduled executions and the start and finish times of each execution. The user can further click each job to inspect the job detail, including the physical plan at each time point, the intermediate states saved by each job, and the early outputs to downstream queries in the pipeline. When the scheduler triggers a job, the UI updates the run-time execution detail of each job. The user can inspect the current execution status of the query, including the amount of early data processed and the amount of early outputs to downstream queries.

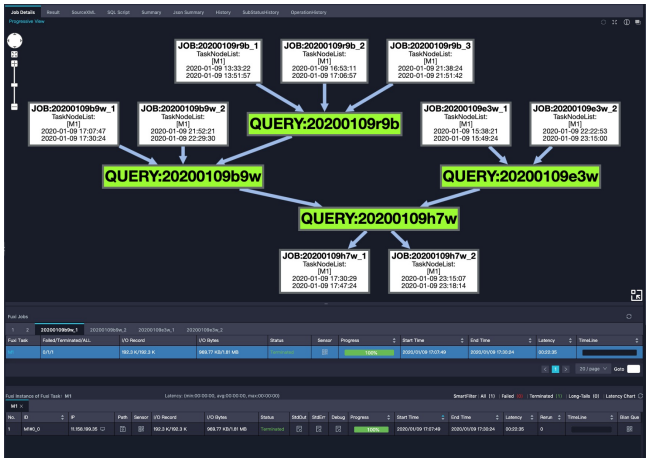

Figure 4: Application UI in Grosbeak

REFERENCES

[1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB* 5, 10 (2012), 968–979.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (2006), 121–142.

[3] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE '95)*. IEEE Computer Society, Washington, DC, USA, 190–200. http://dl.acm.org/citation.cfm?id=645480.655434

[4] Per-Åke Larson and Jingren Zhou. 2007. Efficient Maintenance of Materialized Outer-Join Views. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. 56–65. https://doi.org/10.1109/ICDE.2007.367851

[5] Planning Progressive Execution: Resource Smoothing in Cloud-Scale Data Warehouses [n. d.]. Technical Report, https://kai-zeng.github.io/papers/beanstalk-tech-report.pdf.