

# Evaluation of Load Scheduling Strategies for Real-Time Data Warehouse Environments

Maik Thiele and Wolfgang Lehner

Dresden University of Technology,  
Faculty of Computer Science, Database Technology Group,  
Noethnitzer Str. 46, D-01187 Dresden  
{maik.thiele,wolfgang.lehner}@tu-dresden.de  
<http://wwwdb.inf.tu-dresden.de>

**Abstract.** The demand for so-called living or real-time data warehouses is increasing in many application areas, including manufacturing, event monitoring and telecommunications. In fields like these, users normally expect short response times for their queries and high freshness for the requested data. However, it is truly challenging to meet both requirements at the same time because of the continuous flow of write-only updates and read-only queries as well as the latency caused by arbitrarily complex ETL processes. To optimize the update flow in terms of data freshness maximization and load minimization, we propose two algorithms — local and global scheduling — that operate on the basis of different system information. We want to discuss the benefits and drawbacks of both approaches in detail and derive recommendations regarding the optimal scheduling strategy for any given system setup and workload.

**Key words:** Real-Time Data Warehouse, ETL, Scheduling

## 1 Introduction

Data warehousing and business intelligence have enjoyed immense popularity and success over the last years and now play a key role in strategic corporate decision-making. This evolution raised the need for more up-to-date, so-called real-time, analyses. The real-time aspect in the context of DWHs describes a new processing model where every change is automatically captured and pushed into the DWH. Thus, the data in a real-time DWH is subject to continuous changes, denoted as a trickle-feed of updates. This induces two options from the user's point of view: 1) outdated or slightly outdated data may be used in order to get faster query results, or 2) only the most current data shall be used, i.e., all modifications are committed before the next query is executed. Given that users specify their requirements for each query, we can exploit this information and build a scheduler that controls the continuous update flow [1].

Data changes in DWHs are propagated via ETL processes, which share many similarities with classic manufacturing processes: The data is successively refined

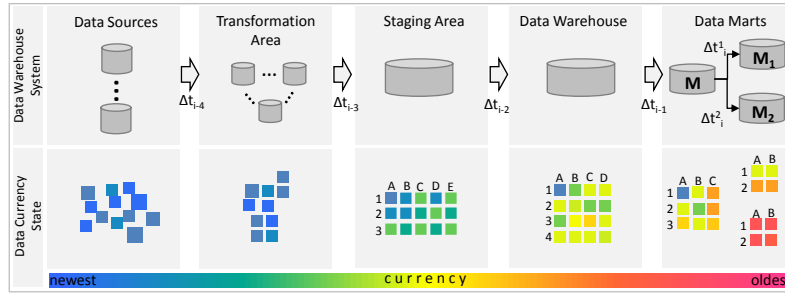


Fig. 1: Push-Based Load of a Data Warehouse and Illustration of Currency

in a multi-stage process, beginning at the data sources, and is finally released to the data marts, where it is analyzed further (Figure 1, right). The multi-stage character of the data production process and the loose coupling of the individual process tasks lead to the observation that — at any given point in time — every data stage illustrated in Figure 1 shows a different degree of currency ( $\Delta t_i$ ). Starting from the data warehouse, the data’s degree of currency increases in the direction of the data sources but decreases in the direction of the data marts (coded with different colors in Figure 1, bottom). These differences in the data’s state also apply to replicas, for example, those used on the level of the data marts (see data marts  $M_1$  and  $M_2$  in Figure 1).

*Opportunities for Improvement:* The push-based data propagation on the one hand and the specification of user requirements on the other hand result in an asynchronous data production process triggered by user requirements. In [2], we presented a scheduling algorithm that prioritizes a set of updates relative to their benefit for concurrent queries. The result was a minimal-intrusive scheduling approach that only propagates as many updates as required by the system users; hence, there is only minimal delay for the execution of the query workload. Our previous considerations were restricted to one database though. However, a data warehouse environment consists of multiple databases that are linked together through a common data production process as described above.

One approach to handle this multi-stage load architecture is to implement a set of schedulers that act locally and independently of the other databases. In the following, we will call this approach *local scheduling*. We want to analyze how this approach performs in comparison to a *global scheduling* that is aware of workload details in all stages and databases respectively. In this paper, we investigate the trade-offs involved with these two approaches.

*Contributions:* In detail, our main contribution comprises the following:

- We formalize and abstract the query processing and update propagation in data warehouse systems. Based on this, we design a model that can be used

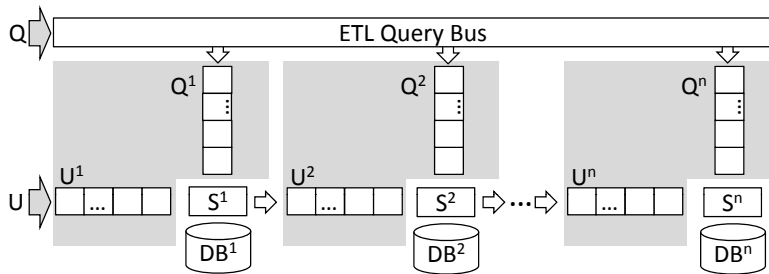


Fig. 2: System Architecture

for analyzing all influential factors relevant to the propagation and scheduling process.

- We propose two scheduling strategies for updates within a distributed real-time data warehouse landscape.
- We evaluate these strategies, generalize our results, and provide some rules of thumb for how to decide on the most useful scheduling strategy in any given scenario.

*Structure of the Paper:* The paper is organized as follows. Our system model and problem statement are outlined in Section 2. We introduce the scheduling algorithms to be evaluated in Section 3. Next, in Section 4, we describe the experimental setup, present our experimental results, and provide scheduling guidelines. Section 5 surveys related work. Finally, we conclude in Section 6.

## 2 System Model and Problem Statement

In the subsequent sections, we will consider our system architecture, the workload parameters as well as the scheduling objectives.

### 2.1 System Architecture

The basis of our analyses is given by a data warehouse environment consisting of a number of decoupled databases that are involved in the same data production process. Queues are conceptually used to link different stages in the production process. There exist two types of queues: one for queries and one for updates ( $Q^i$  and  $U^i$ ). All queries to the data warehouse are distributed to the respective database query queues via a common middleware, the so-called ETL Query Bus. The distribution of the queries to the databases and their scheduling are not part of this paper, but we exploit existing scheduling approaches [3, 4] and assume a pre-defined query order provided by respective query generators. Updates are propagated to the first stage of the data production process. Their processing

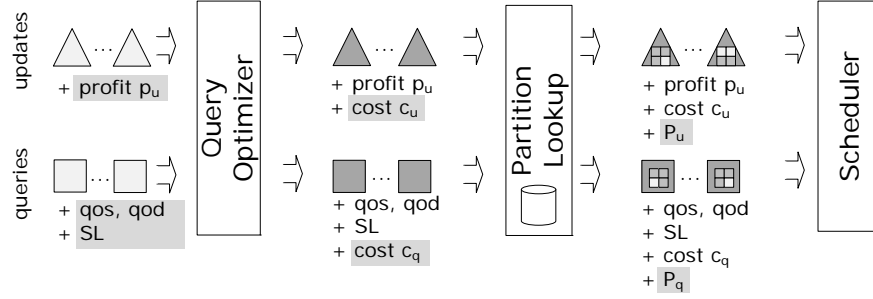


Fig. 3: Workload Model

order is determined by the scheduler, of which we find one per stage  $S^i$  (see Figure 2). Furthermore, the schedulers decide whether it is the query or the update queue that will be served next. The information required for this purpose and its acquisition will be described in the subsequent section. Scheduling can be performed locally or globally. In case of the former, scheduler  $S^i$  only has knowledge of its own stage  $i$ ; in case of the latter, the scheduler disposes of information on the states of all subsequent stages.

We assume that the data production process does not include any branches, as shown in Figure 2. Instead, the updates are propagated linearly from one stage to the next. Such simplification is valid for the purpose of analyzing the scheduler granularity, as we do in this paper.

## 2.2 Workload Model

The workload  $W$  consists of two transaction types: read-only user queries and write-only updates, i.e., an insert, delete, or update. Mixed transactions do not occur, since the push-based DWH approach implies that both queries and updates are submitted independently to the system.

First of all, to apply the scheduling algorithms proposed in this paper, queries and updates are associated with a set of parameters that are derived in a pre-processing step illustrated in Figure 3. Each query  $q_i$  is annotated by a pair  $\langle qos_{q_i}, qod_{q_i} \rangle$ , which specifies the preferences of the user who issued the query (with  $qos_{q_i} \in [0, 1]$  and  $qos_{q_i} + qod_{q_i} = 1$ ). A higher value for  $qos_{q_i}$  denotes a higher demand for QoS (e.g., low response times), whereas a higher value for  $qod_j$  stands for a higher QoD demand (e.g., few unapplied updates). Each update  $u_j$  has a profit parameter  $p_{u_j}$ , which specifies the user benefit if the update is applied. The profit depends on the respective application and can often be calculated easily (e.g., the age of an update or the number of tuples to be inserted, etc.).

In addition, queries are classified into different service levels (SL), which are represented by integer values. On the one hand, a high SL value,  $sl_{q_i}$ , assures that a query  $q_i$  will receive prioritized treatment during query processing (which

is not the focus of this paper). On the other hand, updates correlating with this query  $q_i$  will see stronger prioritization as well.

In order to estimate the execution time of queries and updates, both need to be compiled, which results in a parameter cost  $cost_{q_i}$  for each query and  $cost_{u_j}$  for each update  $u_j$ . The compiled query plans are directly used for their later execution. Updates are assumed to be independent from each other to keep the scenario simple. However, our approach can be easily extended by update execution orders to implement transactional dependencies.

Furthermore, the dependencies between queries and updates must be identified to determine which query profits from which update. Therefore, we assume that the data warehouse is divided into a set of partitions. The set of partitions that are accessed by a query or an update can be determined via an efficient lookup. There exists a dependency between a query  $q_i$  and an update  $u_j$  if the partition sets of both overlap with one another ( $P_{q_i} \cap P_{u_j} \neq \emptyset$ ). A closer look at different partitioning models and their impact on the scheduling quality can be found in [5, 1].

In this paper, we build on the parameters and assumptions described above and focus on the issue of scheduling with different granularities.

### 2.3 Scheduling Performance Objective

Scheduling in the context of real-time data warehouses requires optimization with regard to potentially many criteria: the term real-time represents the demand for updated data, whereas the application of data warehouses implies the desire for fast response times. The subject of this paper’s analyses is the data freshness, which we will consider independently from other optimization criteria. In order to determine the data freshness, we use a lag-based approach, i.e., we take the number of unapplied updates ( $uu$ ) to measure the staleness of a query result. Other data-freshness metrics, such as value-distance metrics or time-differential metrics, may be employed as well. The  $uu$  metric of a query  $q^i$ , executed at stage  $i$ , is affected by an update  $u$  if both access the same partition. Therefore, we consider both the update queue of the current stage and those of all previous stages ( $j \leq i$ ):

$$uu(q^i) = \sum_{u \in U^j, j \leq i, P_q \cap P_u \neq \emptyset} 1. \quad (1)$$

The superscript in the operands denotes the number of the respective stage. To evaluate a scheduling algorithm for a workload  $W$ , we make use of two metrics: the average number of unapplied updates (for a rough estimation of the scheduling’s quality) and histograms (for detailed analyses). We consider both for every individual stage. The average number of unapplied updates of a stage  $i$  for a workload  $W$  is computed as follows:

$$AvgUu^i(W) = \frac{1}{|W_q^i|} \sum_{q \in W_q} uu(q). \quad (2)$$

For a more differentiated evaluation of the scheduling algorithms, we map the number of unapplied updates to histograms. For comparative evaluations, we generate as many histograms as there are stages. The histograms' class width is 1, and the number of bins corresponds to the maximum number of unapplied updates amongst all queries in the system. The y-axis specifies the number of queries that had a specific number of unapplied updates. With the help of the general appearance of the curves, the scattering, and the centralization of the histograms, we can then study the impact of the scheduling algorithms.

## 2.4 Problem Statement

As mentioned, the push-based update propagation in real-time DWHs leads to a multi-stage data production process. In order to schedule updates with only a minimum of intrusion, only those updates should be executed whose modifications are relevant to the query workload of the respective stage. This problem has already been solved for a one-stage scenario in [2]. However, the multi-stage character now creates a complete new set of problems. In the one-stage scenario, there is an update delay resulting from the waiting time for concurrent queries and updates as well as from the update's importance to the user. In the multi-stage scenario, we have to consider not only the current stage but all subsequent stages as well. The number of stages is a multiplicative factor here, which causes major problems for scheduling algorithms.

In the following, we will focus on the comparison of a local scheduling with a global scheduling for multi-stage data production processes. Local scheduling means that a scheduler  $S^i$  only disposes of knowledge of its own workload in the query and update queues. With regard to the loosely coupled system of autonomously acting data nodes, this is certainly useful. However, when considered from a global perspective, the scheduling cannot ever be optimal. Therefore, we want to oppose the locally aware scheduler with a globally aware scheduler, i.e.,  $S^i$  disposes of information about all stages  $i..n$ , where  $n$  is the total number of stages. Since we consider the data freshness for ad-hoc queries, our assumption is that we do not have any knowledge on access patterns, and hence, we have to adjust the scheduler in a reactive manner.

## 3 Scheduling Policies

In the literature, we find a variety of scheduling algorithms for a diverse range of optimization criteria [6], but none of them meets our specific requirements. With our focus being on the minimally intrusive update propagation, we developed a non-preemptive online scheduler [2] that always prioritizes only the most important updates with regard to the average user requirements. With this approach, the user requirements will always be met on average, and the delay for queries caused by updates is as minimal as possible. In order to achieve these objectives, we mapped the scheduling to a knapsack problem and turned the updates into

knapsack elements, defined by profit (improvements in data freshness) and costs (delay of queries). The knapsack size is determined through the user requirements. We will skip the details in this paper and focus on the basic properties of this scheduling algorithm now.

### 3.1 Scheduling Algorithms for Push-Based Update Propagation

We will very briefly introduce our scheduling algorithm to be used as a building block for analyzing large data processing chains. Since the objective of our analysis focuses on the scheduling granularity, we will only provide a simplified description of the scheduler itself.

As already sketched in Figure 2, there is one scheduler  $S^i$  for each stage. This respective scheduler is triggered by the execution of either a query or an update. The overall scheduling algorithm can be seen in Figure 4. First, the average of all QoS/QoD values of all queries in the respective queue is computed. Subsequently, a random value is generated to decide whether the query queue or the update queue should be prioritized. For a high QoS value, the likelihood of the query queue being prioritized is higher, and vice versa.

If the query queue is selected, the query with the highest prioritization will be executed. At this point, we take the query schedule for granted and do not look into it any further. Once the query has been executed, Equation (1) will be used to determine the number of updates that correlate with this query and have not yet been propagated.

If the execution of an update is favored, we initially differentiate between two modes: the *local scheduling* and the *global scheduling* of updates. In case of the former, we determine for a stage  $k$  all correlations between queries and updates in the respective queues,  $Q^k$  and  $U^k$ . In case of the global scheduling, we determine the correlation between all queries  $q_i \in \bigcup_{l=k..n} Q^l$  and  $U^k$ , and between all updates  $u_j \in U^k$ . That is to say, we consider not only the queries in the same stage but also the queries of all subsequent stages as well. After all dependencies have been determined, the updates will be scheduled based on their priorities. The priority results from the number of correlations with queries. Additionally, query service levels (see Section 2.2) may be used to weight the correlations accordingly. Effects of such weightings will be examined in Section 4. Finally, the update with the highest priority will be executed. This rather simplified representation of the algorithms is sufficient enough to allow us to take a closer look at the scheduling granularity and its effects on the data freshness, respectively, in the next section.

## 4 Evaluation and Discussion

In this section, we compare the local and global scheduling strategies in different scenarios. Our goal here is to determine the requirements for global scheduling to perform better than local scheduling. Additionally, we quantify this improvement

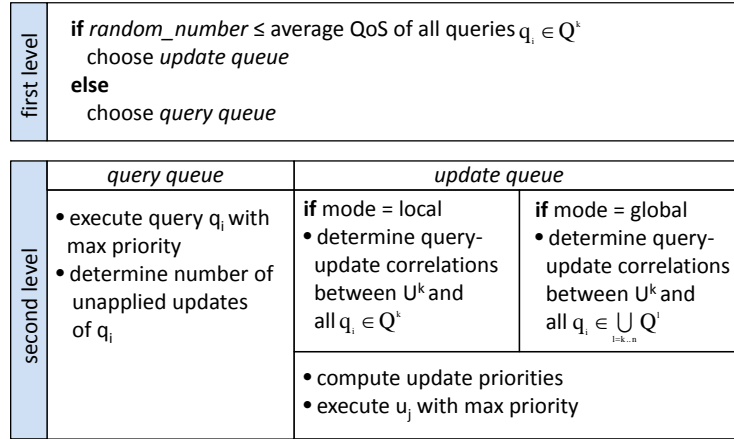


Fig. 4: Scheduling Algorithm Overview

with regard to the data freshness. Based on the results, we enhance the algorithm presented in 3.1 and develop guidelines for when to use which scheduling approach. Each experiment will be summarized in a *lesson learned* paragraph. Since the construction of a truly distributed environment is very consuming and expensive, we developed an appropriate simulation environment. This allows us to compare data warehouse environments of different size with the help of a broad set of experiments.

#### 4.1 Simulation Framework

Our experimental setup consists of a *workload generator* and a *set of data stages*, each comprising a query and an update queue, a scheduler component, and an execution component (see Figure 2). The simulation environment is located on the same machine: an Intel Core 2 Duo 2.5 GHz system running Windows XP with 4 GB of main memory. The queries and updates generated with the workload generator can be varied with regard to different parameters: number of queries and updates, load, user requirements regarding QoS and QoD, and query and update costs (drawn from a Gaussian or a Pareto distribution). Additionally, we can modify the degree of dependency between queries and updates as well as the length of the data production process.

In order to evaluate the scheduling approaches, we use the metrics developed in 2.3: the average number of unapplied updates and their representation as histograms.

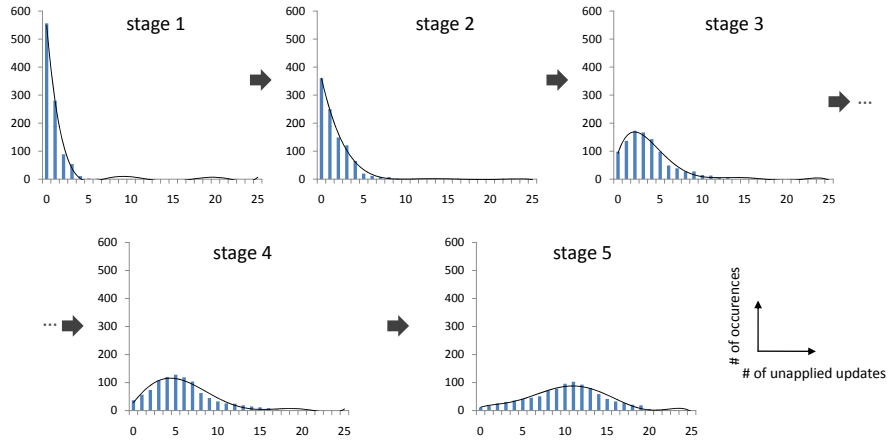


Fig. 5: Number of Unapplied Updates with Increasing Length of the Data Production Process

## 4.2 Effect of the Data Production Process Length

In a first step, we want to analyze the impact of length  $n$  of the data production process and the associated delay for updates. For this purpose, we generated a workload with 2,500 queries and 1,000 updates. The costs (execution times) were drawn from a normal distribution ( $\mu = 100ms$ ,  $\sigma = 20ms$ ). The data production process consists of  $n = 5$  stages. We applied the *local scheduling* and executed the workload 20 times to get valid results. This led us to the five histograms (one per stage) given in Figure 5. The bins denote the number of unapplied updates (uu) that occurred in the workload; the y-axis gives their frequencies. We can see for stage 1 that the majority of queries returned up-to-date results (0 uu), and only few updates could not be propagated in time (up to 5 uu). However, the later a stage takes place during the data production process, the more the data freshness deteriorates. In stage 5, we find queries with up to 20 unapplied updates, with most of them being around 10 uu. The reason for this deterioration of data freshness lies in the increasing delay for updates due to the growing number of stages. Updates that correlate with queries in later stages are blocked by earlier stages, since those base their prioritization only on local user requirements.

**Lesson Learned 1:** *The data freshness decreases with increasing length of the data production process.*

## 4.3 Comparison of Local and Global Scheduling

In the experiment described above, we only applied local scheduling. In a second step, we use the same workload to compare local and global scheduling. For this

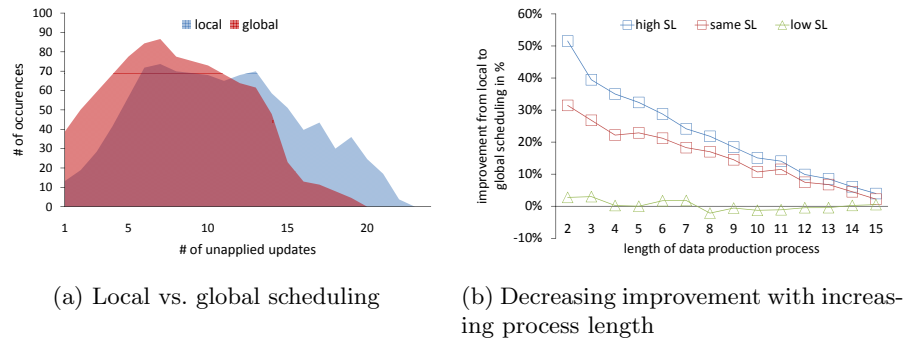
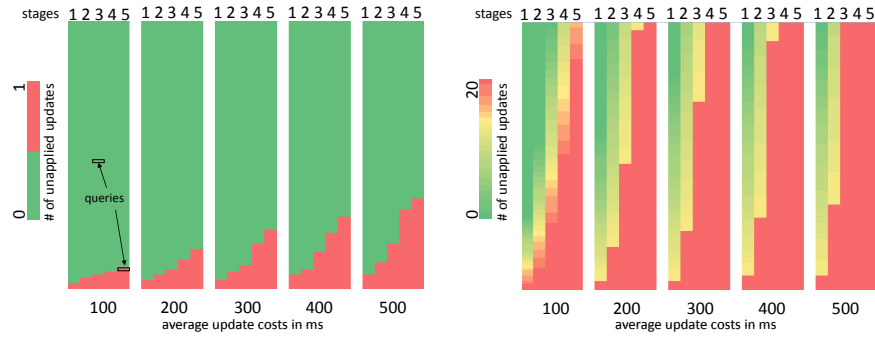


Fig. 6: Comparison of Global and Local Scheduling

purpose, we analyzed the data freshness in stage 5 of a 5-stage process for both approaches. We assigned the highest service level in the whole workload to those queries that are processed in stage 5. The result is shown in Figure 6a. We can see that the distribution of unapplied updates is better for global scheduling than for the local approach. On average, each query had 10.7 uu when using local scheduling and 8.1 uu when using global scheduling, which corresponds to an improvement of about 30%. If we keep in mind that the queries (and hence the correlating updates) had been assigned the highest priority, an improvement of 30% is relatively low. We will study this behavior in more detail in Sections 4.4 and 4.5.

At this point, we study further the effects of the data production process' length. Figure 6b shows the improvement for global scheduling compared to local scheduling based on measurements taken from the  $n$ -th stage of an  $n$ -stage process. We assigned 1) the highest service level to queries in the highest stage (high SL), 2) the same SL to all queries (same SL), and 3) the highest SL to the queries in the first stage (low SL). For a 2-stage process, we find improvements of 50%, 30% and 0%, respectively, for the individual service-level assignments. However, the improvements of global scheduling compared to local scheduling shrink continuously with increasing process length. For a 15-stage process, the advantage in the 15th stage is only 3%. Additionally, the service levels' impact is weakened with increasing process length. In stage 5, the *high SL* outperforms the *same SL* by no more than just 7%. Since the prioritization of a certain stage comes at the cost of other stages' performance, the use of service levels is generally not recommended. The reason is found in the growing delay with longer processes. For example, updates for the 15th stage are handled with priority, but the delay in the preceding 14 stages makes it impossible to propagate them in time.

Both experiments have shown that the benefit to be achieved via global scheduling is limited. The blocking factor is the length of the process. In the following experiments, we want to detect other restrictive factors and derive recommendations on when to use what type of scheduling under given circumstances.



(a) Each query is correlated with 1 update (b) Each query is correlated with 20 updates

Fig. 7: 100 Queries in 5 Stages, Increasing Update Costs

**Lesson Learned 2:** *The global scheduling is only beneficial for relatively short data production processes. Furthermore, the impact of service levels is restricted by the process length.*

#### 4.4 Effects of Stage-Concurrent and Long-Running Updates

In our previous experiments, we assumed query and update costs to be equal. In the next experiment, we analyze the scheduling behavior for varying update costs. The experimental setup as well as the results will be explained by using Figure 7. In five queues or stages, respectively, there are 100 queries each. A queue’s head is at the lower end of the figure. Every query has a color code (cf. the respective legend to the left of the figure). A query’s color represents the number of unapplied updates that will exist in the best possible case upon query execution. The service levels of the query were assigned in ascending order, i.e., queries in stage 1 have the lowest level, and queries in stage 5 have the highest level. The query costs were derived from a normal distribution ( $\mu = 100ms$ ,  $\sigma = 20ms$ ), but the update costs vary ( $\mu = 100ms$  to  $500ms$ ). In our first experiment (see Figure 7a), there is exactly one correlating update for all queries of a stage. That means there are five updates with disjoint correlations, e.g., all queries to stage 1 correlate with an update  $u_1$ , all queries to stage 2 correlate with  $u_2$  but not with  $u_1$ , etc. If we raise the update costs from 100ms to 500ms, we will see that for an increasing number of queries (marked in red) the correlating update cannot be propagated in time. In our concrete example with update costs of 500ms in stage 5, this concerns approximately a third of all queries.

In our second experiment, each query of a stage correlates with 20 updates, which are again disjoint from stage to stage. We raise the update costs from 100ms to 500ms again. As we can see in stage 5, for update costs larger than 100ms, all queries will always have the lowest data freshness upon their execution, i.e., 20

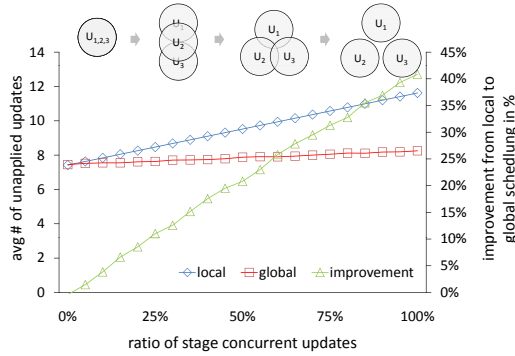


Fig. 8: Varying Ratio of Stage-Concurrent Updates

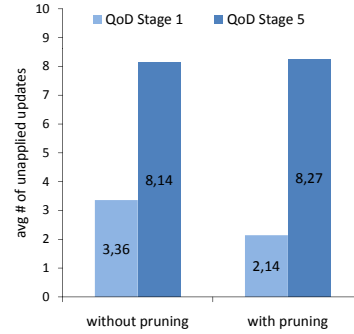


Fig. 9: Effect of Pruning in the Global Scheduling

unapplied updates. For update costs of 500ms, this even applies to stages 3 to 5.

To summarize, it is impossible to improve data freshness for any of the red queries shown in Figures 7a and 7b without causing additional delay for these queries. We denote this type of queries as *irretrievable queries*. The number of *irretrievable queries* increases 1) with the deteriorating cost ratio between queries and updates, 2) with a growing number of updates that are disjoint from stage to stage, and 3) with the increasing length of the data production process, as shown in Section 4.3. The data freshness of the *irretrievable queries* is invariant to the selected scheduling approach for updates. Thus, global scheduling would not improve the data freshness of these updates any further. In Section 4.6, we will look at this aspect in more detail by evicting the *irretrievable queries* in the global scheduling.

**Lesson Learned 3:** *With an increasing number of irretrievable queries, the global scheduling deteriorates compared to the local scheduling.*

#### 4.5 Ratio of Stage-Concurrent Updates

The aspect of stage-concurrent updates should be considered in another experiment. For this purpose, we used equal costs for queries and updates, as we did in Section 4.2, but we varied the amount of overlaps and disjointness of updates, respectively. Figure 8 gives a symbolic representation with Venn diagrams. For a ratio of 0% (x-axis), the queries of all stages correlate with the same set of updates. The disjointness ratio increases in steps of 5% until finally the queries of each stage correlate with disjoint sets of updates each. Again, the length of the data production process is  $n = 5$ . The queries to stage 5 had been assigned the highest service level. For each grading of the disjointness, we applied local and global scheduling and measured the average number of unapplied updates for

stage 5 (see Figure 8). For a disjointness ratio of 0%, both scheduling approaches deliver the same result. Since the query correlations are identical for all stages, the global approach does not generate any extra benefit. However, with increasing disjointness ratio, the global scheduling successively improves compared to the local approach. For totally disjoint update sets, the global scheduling reaches a 40% improvement compared to local scheduling.

**Lesson Learned 4:** *The consideration of all query-update correlations within the data production process is only beneficial when there are minimally overlapping or completely disjoint update sets.*

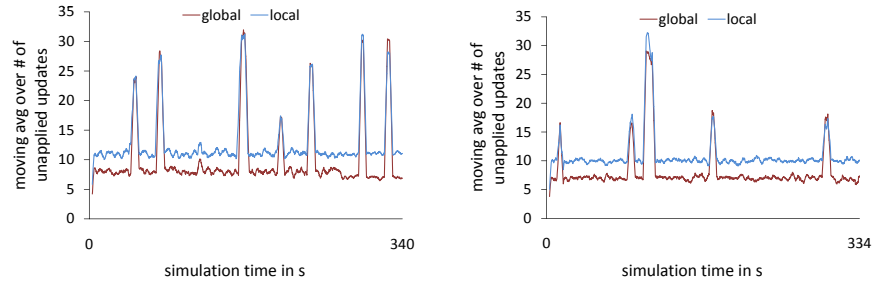
#### 4.6 Pruning of Irretrievable Queries

In Section 4.4, we showed that it is impossible to improve the data freshness for the so-called *irretrievable queries*. We now want to use this fact to optimize the global scheduling. For this purpose, correlations with these query types will remain unconsidered when prioritizing updates. This decreases the number of updates that receive high priority but cannot be propagated in time due to the delay described in 4.2. As a result, the load decreases in all stages to which these updates will be propagated. In order to evaluate the effectiveness of this approach, we measured the data freshness in stages 1 and 5 of a 5-stage process both with and without this optimization (see Figure 9). The experimental setup is the same as in Section 4.3. As we can see, the optimization only slightly decreased the data freshness in stage 5 but improved it by 60% in stage 1.

**Lesson Learned 5:** *The effect that the improved data freshness in stage  $j$  leads to a deterioration of freshness in all stages before  $j$  can be weakened by pruning irretrievable queries.*

#### 4.7 Effects of Long-Running Update and Queries During Runtime

We saw in 4.4 that a low ratio of query and update costs often results in *irretrievable queries*, which comes with negative effects on the global scheduling. In this experiment, we want to analyze the runtime behavior for workloads with long-running queries and updates. In a first experiment, we drew the costs for the 2,500 queries from a normal distribution ( $\mu = 100ms$ ,  $\sigma = 20ms$ ), and the costs for the 1,000 updates were derived from a Pareto distribution with a minimum value of 50ms and an alpha value of 0.8; the maximum costs were limited to 5,000ms. Figure 10a shows the result over the whole simulation time. We averaged the number of unapplied updates over a window size of 25 queries. For a balanced query-update cost ratio, the global scheduling performs better than the local scheduling, as the previous experiments confirmed. However, if long-running updates are executed, both the local and the global scheduling deteriorate to the same extent and slowly converge. The reason is that the execution of a long-running update significantly delays the update propagation and thereby disrupts the temporal locality. Any consideration of correlations over all



(a) Pareto-Distributed Update Costs

(b) Pareto-Distributed Query Costs

Fig. 10: Effect of Long-running Updates and Queries

stages is thus obsolete in these cases.

In a second analysis, we used a setup analog to the first one but derived the update costs from a normal distribution and the query costs from a Pareto distribution. The result is shown in Figure 10b. On average, the global scheduling slightly outperforms the local scheduling again. However, as soon as we execute long-running queries, we get the same result as above. The long-running queries block the data production process, which disrupts the temporal locality between a query and its correlated update set.

**Lesson Learned 6:** *With an increasing number of long-running queries and updates, the global scheduling deteriorates compared to the local scheduling.*

## 5 Related Work

The subject of scheduling algorithms has been discussed extensively in the research community. Since there are no algorithms for our problem — i.e., the scheduling of updates in real-time DWHs — we only refer to [6] at this point though. Our update prioritization shares some similarities with the transaction scheduling techniques in real-time database systems [7, 8, 9, 10]. These approaches often work with deadline semantics, where a transaction only adds value to the system if it finishes before its deadline expires. For this purpose, the DBA of a system specifies the acceptable miss ratio threshold, i.e., the DBA defines the number of transactions that may be aborted without negatively affecting the functionality of the system. The abortion of transactions is necessary to ensure that guarantees in terms of the system’s real-time properties will be met. However, real-time in our context refers to the insertion of updates that happens as quickly as possible or as quickly as needed, respectively, depending on the user requirements. Nevertheless, the approaches found in several algorithms from the field of real-time databases are still interesting for this paper. Heavy-tailed workloads, where a small number of long-running queries sign responsible for the majority of the load, are typical for data warehouse and BI

applications. The identification and classification of long-running queries was analyzed in [3]. A scheduler for heavy-tailed workloads that is based on the stretch metric was introduced in [4]. In this paper, we examined the influence of long-running queries and updates on the data freshness. In one of our previous works ([2]), we designed an online scheduling algorithm that prioritizes updates with regard to their profit and costs. That scheduler inherently handled long-running updates.

In [11], the authors propose an ETL quality metric suite that aims to handle these metrics on all the ETL design levels. The paper raises awareness for the different relationships among the metrics and the resulting trade-offs between alternative optimizations of ETL processes. Our work addresses two specific metrics: query performance and data freshness; we give recommendations with regard to the scheduling and coupling of individual nodes of an ETL process.

Research activities similar to ours can be found in the field of multi-tier cache management [12, 13, 14]. The problem there is that the temporal locality continuously decreases from the first-tier cache down to the lower caches. There are two collaborative approaches in order to increase the cache hit ratio: hint-based [15] and client-controlled [16] caching schemes. Furthermore, transparency is traded in for the possibility of improved performance there as well. The difference, however, lies in the access structures: for multi-tier cache management, we find a strict top-down structure, whereas our real-time data warehouse architecture allows us to route queries to any arbitrary node.

## 6 Conclusion

In this paper, we looked at data production processes in real-time DWH environments and discussed the granularity level of scheduling algorithms. We defined local and global scheduling as two extreme cases and analyzed and compared them with a variety of experiments. Surprisingly, the global approach often only slightly outperforms the scheduling approach based on local information. An exception is only found if we use very extreme assumptions; in that case, the global scheduling delivered a significant benefit. In general, we therefore recommend using individual schedulers for each stage, which corresponds to the loose coupling that characterizes ETL and data production processes, respectively. Particularly for larger processes, we do not recommend evaluating query-update correlations globally. However, a hybrid approach — i.e., the partial scheduling over sub-sets of stages — may prove useful under certain conditions.

## References

1. Thiele, M., Fischer, U., Lehner, W.: Partition-based workload scheduling in living data warehouse environments. *Information Systems* **34** (2009) 382–399
2. Thiele, M., Bader, A., Lehner, W.: Multi-objective scheduling for real-time data warehouses. In: *In Proceedings der 12. GI-Fachtagung für Datenbanksysteme in Business, Technology und Web, GI* (2009) 307–326

3. Krompass, S., Kuno, H., Wiener, J.L., Wilkinson, K., Dayal, U., Kemper, A.: Managing long-running queries. In: EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology, New York, NY, USA, ACM (2009) 132–143
4. Gupta, C., Mehta, A., Wang, S., Dayal, U.: Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. In: EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology, New York, NY, USA, ACM (2009) 696–707
5. Thiele, M., Fischer, U., Lehner, W.: Partition-based workload scheduling in living data warehouse environments. In: DOLAP, New York, NY, USA, ACM (2007) 57–64
6. Leung, J., Kelly, L., Anderson, J.H.: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press, Inc., Boca Raton, FL, USA (2004)
7. Kang, K.D.: Managing deadline miss ratio and sensor data freshness in real-time databases. TKDE **16**(10) (2004) 1200–1216 Senior Member-Sang H. Son and Fellow-John A. Stankovic.
8. Kang, K.D., Son, S.H., Stankovic, J.A., Abdelzaher, T.F.: A qos-sensitive approach for timeliness and freshness guarantees in real-time databases. In: ECRTS. (2002) 203–212
9. Haritsa, J.R., Carey, M.J., Livny, M.: Value-based scheduling in real-time database systems. The VLDB Journal **2**(2) (1993) 117–152
10. Hong, D., Johnson, T., Chakravarthy, S.: Real-time transaction scheduling: A cost conscious approach. In Buneman, P., Jajodia, S., eds.: SIGMOD, ACM Press (1993) 197–206
11. Simitsis, A., Wilkinson, K., Castellanos, M., Dayal, U.: Qox-driven etl design: Reducing the cost of etl consulting engagements. In: Appears in SIGMOD'09: International Conference on Management of Data, New York, NY, USA, ACM (2009)
12. Zhou, Y., Chen, Z., Li, K.: Second-level buffer cache management. IEEE Trans. Parallel Distrib. Syst. **15**(6) (2004) 505–519
13. Gill, B.S.: On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In: FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, USENIX Association (2008) 1–17
14. Chen, Z., Zhang, Y., Zhou, Y., Scott, H., Schiefer, B.: Empirical evaluation of multi-level buffer cache collaboration for storage systems. In: SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, New York, NY, USA, ACM (2005) 145–156
15. Li, X., Aboulnaga, A., Salem, K., Sachedina, A., Gao, S.: Second-tier cache management using write hints. In: FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, USENIX Association (2005) 9–9
16. Wong, T.M., Wilkes, J.: My cache or yours? making storage more exclusive. In: ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2002) 161–175