Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /
This is a self-archiving document (accepted version):

Maik Thiele, Ulrike Fischer, Wolfgang Lehner

Partition-based workload scheduling in living data warehouse environments

Erstveröffentlichung in / First published in:

Information Systems. 2009. 34(4-5), S. 382-399. Elsevier. ISSN 0306-4379.

DOI: https://doi.org/10.1016/j.is.2008.06.001

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-863767

SLUB
Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA
Quality Content of Saxony

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

Partition-based workload scheduling in living data warehouse environments

Maik Thiele *, Ulrike Fischer, Wolfgang Lehner

Dresden University of Technology, 01062 Dresden, Germany

ARTICLE INFO

Keywords:
Scheduling
Real-time data warehouse
Resource allocation

ABSTRACT

The demand for so called living or real time data warehouses is increasing in many application areas such as manufacturing, event monitoring and telecommunications. In these fields, users normally expect short response times for their queries and high freshness for the requested data. However, meeting these fundamental requirements is challenging due to the high loads and the continuous flow of write only updates and read only queries that might be in conflict with each other. Therefore, we present the concept of workload balancing by election (WINE), which allows users to express their individual demands on the quality of service and the quality of data, respectively. WINE exploits these information to balance and prioritize both types of transactions—queries and updates—according to the varying user needs. A simulation study shows that our proposed algorithm outperforms competing baseline algorithms over the entire spectrum of workloads and user requirements.

1. Introduction

The high popularity and success of data warehousing as well as the need for up to date information within organizations have led to the new demand for so called living or real time data warehouse environments. In the context of data warehouses, real time means that every time a change occurs in the modeled environment, it is automatically captured and pushed into the data warehouse. This is different to the traditional pull based batch ETL processing, where users have to specify the data they want to see. The trend towards real time data warehouses is enforced by the increasing number of globally oriented companies, which are characterized by their heterogeneous structure that results in the temporally dispersed production of data; this makes it almost impossible to find a suitable point in time to load the warehouse. The push based data integration leads to a continuous stream of write only updates, which compete for system resources with read only queries from the user side (Fig. 1). This paper addresses the problem of how to schedule the two conflicting types of transactions in order to satisfy the user requirements; these are outlined in the following.

In applications on top of a data warehouse, such as OLAP or data mining tools, users expect short response times for their queries (quality of service, QoS) and high freshness of data, i.e., they want the number of unapplied updates for their query results to be as low as possible (quality of data, QoD). Ideally, if enough computing resources are available, both needs can be satisfied, i.e., the highest data freshness associated with updates and the shortest response times associated with queries can be guaranteed. Due to periods of high load in the data warehouse or unpredictable data request and update patterns, however, it may be very hard to comply with both requirements at the same time. To tackle the problem of allocating system resources in the most

* Corresponding author. Tel.: +49 35146338283; fax: +49 35146338259.
E-mail addresses: maik.thiele@tu-dresden.de, maik.thiele@inf.tu-dresden.de (M. Thiele), ulrike.fischer@tu-dresden.de (U. Fischer), wolfgang.lehner@tu-dresden.de (W. Lehner).

doi:10.1016/j.is.2008.06.001

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

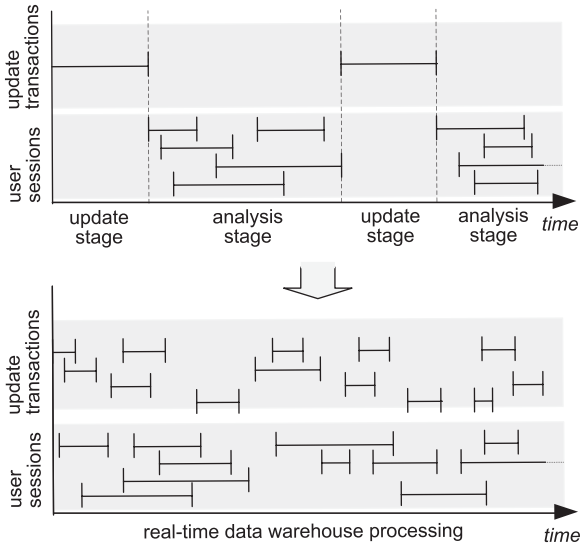

Fig. 1. From traditional ETL processing to real-time data warehousing.

efficient way and to find a reasonable balance between both needs, the order in which queries and updates are executed needs to be scheduled under consideration of the user's preferences. We believe that some users might be willing to accept a certain degree of staleness as a trade off for very fast answers, whereas other users may prefer a high degree of freshness that comes with longer execution times. This trade off between QoS and QoD needs to be specified by the user for all queries and will be applied by the underlying system to control the query and update flow in an appropriate manner. Thus, the user requirements specified for each query (QoS or QoD) can be seen as a simple vote to prioritize either queries (*query mode*) or updates (*update mode*).

We claim that there are certain visible trends in the behavior of user groups. For example, the majority of users will prefer fast but less up to date responses during their everyday work, whereas at the end of the month, when business reports need to be drafted, the same users will accept slower response times in favor of up to date responses.

1.1. Example

We will sketch our main principles with the help of an example as illustrated in Fig. 2. To balance between the usually independent transactions, both the update trans actions from the staging area and the user queries need to be coupled through a common middleware, called the *workload balancing unit* (WBU). The WBU is responsible for the global scheduling; thus, it allocates resources either to the query or the update queue, depending on the user demands. In our example, the WBU would allocate the underlying data warehouse to the query queue, since the sum of QoS values exceeds the sum of QoD values, i.e., on average, the users favor short response times. At a lower level, queries and updates can be prioritized according to the user demands. For example, query q_1

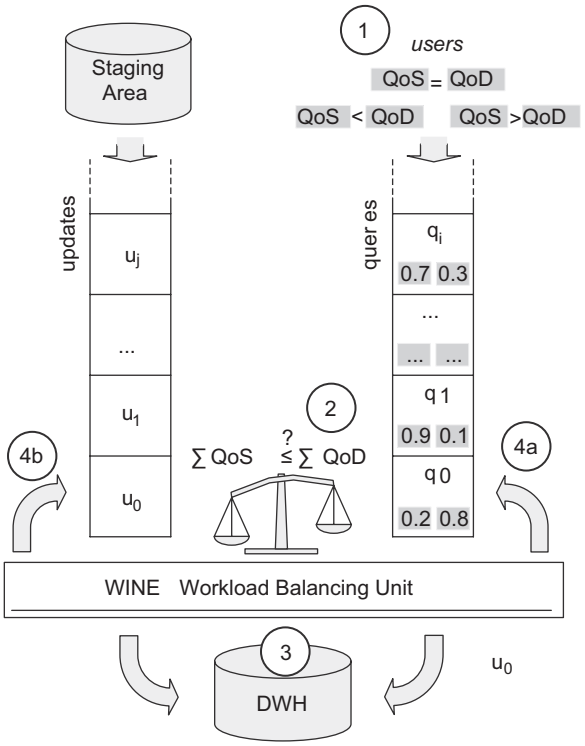

Fig. 2. WINE workload balancing system.

would be executed before query q_0, since the QoS value of q_1 is greater than the QoS value of q_0. Similarly, we propose a prioritization algorithm for updates. So, our overall optimization goal is to minimize the retention time for queries with high currency priorities (QoS) and to maximize the freshness for queries with high freshness demands (QoD). In contrast to the response time, which denotes the overall time for a query execution, the retention time just denotes the time the query has to wait for system resources and it should be minimized by our scheduling algorithm.

1.2. Contributions

Our main contributions are:

- We identify different user groups regarding their demands for data freshness and query retention times, respectively.
- We propose a partitioning scheme to find correlations between queries and updates, i.e., to find those queries and updates that access the same data items.
- We utilize the user preferences and develop a two level scheduling algorithm:
  - First, we balance over the query and update queues (steps 2 and 3 in Fig. 2).
  - Second, queries and updates are prioritized con cerning the QoS and QoD values, respectively (steps 4a and 4b in Fig. 2).

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

To illustrate the impact of our scheduling algorithm, we conducted a set of experiments using synthetic and real world datasets. We simulate the traditional transaction processing, which handles query and update transactions independent of each other, using three common schedul ing policies:

(1) First In, First Out (FIFO [1]): This is to simulate a schedule where queries and updates are executed according to their arrival times. FIFO is a special case of our scheduling algorithm and occurs when all users abstain from voting, i.e., when they assign the same weight to QoS and QoD.
(2) FIFO Query High (FIFO QH [2]): This consists of two queues: one for queries and one for updates. The FIFO query queue has a higher priority than the FIFO update queue. Thus, FIFO QH achieves the highest QoS.
(3) FIFO Update High (FIFO UH FIFO QH [2]): This con sists of two queues similar to FIFO QH, and the FIFO update queue has a higher priority than the FIFO query queue. Thus, FIFO UH performs best regarding QoD.

1.3. Structure of this paper

The paper is organized as follows. Section 2 provides a review of related work and describes the requirements our scheduling algorithm has to meet. We outline our system structure in Section 3. We introduce WINE, our two level scheduling algorithm, in Section 4. Next, in Section 5, we describe the experimental setup and present our experimental results. Finally, we conclude in Section 6 and discuss topics of future research.

2. Related work

The terms *active data warehouse* and *real time data warehouse* are closely linked and unfortunately often erroneously used as synonyms in the database literature. The concept of active data warehousing refers to a paradigm where the warehouse itself actively reacts to occurring events. That is to say, analyses previously performed by using OLAP are now automated and processed by the data warehouse itself. This technique is usually implemented by employing so called *event, con dition, action* (ECA) rules [3]. That means active data warehousing represents the change from a pull based architecture to a push based architecture.

In contrast to that, the goal of real time data ware houses, also known as zero latency or living data ware houses, is to minimize the latency time, i.e., the time between the event in the real world and the moment that the data record becomes available for analyses in the data warehouse. That is to say, the term real time in this context refers to the fast addition of data without the need to obey defined time limits.

Often, the argument is brought up that real time data warehouses are obsolete, since the requirements imposed on them are already addressed by operational data stores (ODSs). Both are subject oriented and include integrated data with the goal to reduce latency times. Despite its very low latency time, however, the real time data warehouse has to be classified as an analytical system, whereas the ODS resulted from the demand for the integration of operational data. That is to say, the data stock of an ODS is volatile, while the stock of a data warehouse is not.

The subject of scheduling algorithms has been dis cussed extensively in the research community and thus, a variety of works exist. However, at this point, we only refer to [1]. Scheduling algorithms are often classified as online or offline and as preemptive or non preemptive algorithms. In this paper, we focus on online and non preemptive scheduling. That is to say, query attributes such as, for example, the estimated execution time, are not known in advance and we do not interrupt any queries or updates that are in the process of being executed. In order to test the scheduling features of WINE, we there fore only use algorithms of the same class (FIFO, FIFO QH and FIFO UH) for our comparisons. The scheduling algorithms FIFO QH and FIFO UH, which prefer queries or updates, respectively, are slightly modified versions of the algorithms found in [2], since we do not employ an economic model. Scheduling algorithms such as, for example, shortest job first (SJF), which is known to be optimal for minimizing the mean response time (corre sponds to our QoS metric), have not been considered for this paper, since the application of these algorithms necessitates a priori knowledge of the execution time.

Economic based resource allocation and pricing schemes have been considered by many recent research ers: Fergison et al. [4,5], Kurose et al. [6], Stonebraker et al. [7] and Carl et al. [8,9]. However, in most cases similar to our data warehouse scenario, it is impossible for the user to determine the valency of a specific QOD or QoS criterion; it is particularly difficult to weigh the different service criteria by assigning varying values. Additionally, there is a lack of accounting models for computer systems in general and for the OLAP world in particular.

The idea of an economic approach was picked up by Qu et al. [2,10] and assigned to the problem of workload balancing in data intensive Web applications. To max imize the overall system profit, they proposed an adaptive algorithm called QUTS, where users are able to indicate their preferences by expressing the potential economic benefit of given queries through appropriate values for their quality criteria, i.e., for different levels of query deadlines and unapplied updates. Even though their work is the closest in spirit to ours, there are still several significant problems and differences: First, there is a lack of economic approaches for users of databases in general, as mentioned above. Second, it is not possible for users of databases and OLAP tools to specify query deadlines; and neither are they able to decide on the number of unapplied updates they are willing to accept. A solution would be to enrich the user interface by additional information, such as average query deadlines, the actual load of the system, and the number of unapplied updates on data items of interest. Nevertheless, due to complex system components, such as buffer management and

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

I/O scheduling, it is very hard to predict query runtimes and to decide for a specific query deadline, respectively. Qu et al. circumvented this problem by focusing on main memory databases only, i.e., they reduced the problem to CPU scheduling.

For these reasons, we provide users with a simple scale to express their preferences; hence, they explicitly vote for a system mode that either supports freshness by prefer ring updates or that provides fast query processing by focussing less on the query freshness. Instead of following approaches that process queries in order of their economic benefit, we propose a democratic system, where each vote has the same weight, which avoids the problems of economic approaches mentioned above.

Yet another difference to QUTS can be found in the granularity of queries and updates. In the Web database environment, they are only processed one tuple at a time, which makes the correlation very easy, whereas in data warehouses, they are processed on an arbitrary part of the multi dimensional dataset. In order to ensure a correla tion between queries and updates anyway, we divide the multi dimensional data into comparable partitions.

Our query prioritization shares some similarities with the transaction scheduling techniques in real time data base systems [11 14]. 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 real time properties of the system 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 require ments. Nevertheless, the approaches found in several algorithms from the field of real time databases are still interesting for this paper.

The data warehouse maintenance process, i.e., the insertion of updates, can be split into two phases: (1) The external maintenance phase denotes the main tenance process between the information sources (IS) and the warehouse or its base tables. (2) The external maintenance phase refers to the process of maintaining materialized views with the base tables used as founda tion. In this paper, we focus on the first phase and initially assume a model with a single queue and a single thread. That is to say, updates are inserted sequentially and in order of their importance for the query side. The maintenance of materialized views and the various aspects of this discipline, such as incremental mainte nance or concurrent updates [15 20], are not in the center of attention of this paper. On the other hand, the partitioning schema proposed here may also be used to detect correlations between queries and updates in order to spot conflicts of concurrent updates when maintaining the materialized views.

For replicated and cached data, the authors of [21,22] presented a model for currency and consistency con straints in queries and developed techniques to allow the DBMS to guarantee compliance with these constraints. Given explicit user constraints, as in our approach, the DBMS tries to find the best routing decision (to replicates of different freshness) for each query in a consistent manner. This is similar to WINE, where quality constraints are used to decide whether or not updates or queries need to be preferred and which of them deserve prioritization.

Since the deferred insertion of updates is demand driven, it may occur that at a given point in time, several updates exist in the waiting queue that all need to access the same data object. In case these modifications mutually neutralize themselves, these redundant updates may be removed from the queue, thus saving valuable system resources. In [23], a condensed operator is presented in the context of maintenance of materialized views. A similar operator could also work for our application but we will not go into detail here. Also, it might be possible to group updates, as it is done with many view maintenance algorithms [15,24,25].

The vertical and horizontal partitioning of databases, with special consideration given to the optimization of performance, is addressed in [26]. They propose an algorithm that analyzes the workload and generates interesting column groups. In a slightly different form, this algorithm can also be used for the candidate generation of the partitioning attributes. The partitioning of data in so called cubelets to determine the complete ness of aggregates was the subject of one of our earlier papers [27].

3. System structure

As previously mentioned, we assume a real time data warehouse environment. Our scenario setup consists of the following three components: (1) a central RDBS to represent the data warehouse, (2) a staging area to provide a permanent stream of updates trickling into the data warehouse, and (3) some application on top of the data warehouse to feed it with user defined queries. Further infrastructure components such as aggregate tables or data marts are omitted due to limited space.

3.1. Workload model

In our scenario, the workload W consists of two kinds of transactions: read only user queries $q_i \in W_q$ and write only updates $u_j \in W_u$, where $W = W_q \cup W_u$ (Table 1). Mixed transactions do not occur, since the push based approach means that both queries and updates are submitted independently to the system and added to the query queue Q and the update queue U, respectively. The set of all user queries in the query queue is denoted as $Q = \{q_i | i \geqslant 0\}$; the set of all updates in the update queue is $U = \{u_j | j \geqslant 0\}$. The length of the queues increases or decreases with the number of incoming or processed transactions, respectively. Potential overflows caused either by very high arrival rates of transactions or by a very low system throughput cannot be considered here, since these problems can only be solved via an

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

Table 1
Notation table

Symbol	Description
W	Workload to be processed: $W \quad W_q \cup W_u$, where W_q denotes the set of queries and W_u is the set of updates in the workload
Q	Query queue of length $\|Q\|$
U	Update queue of length $\|U\|$
pos_{q_i}/ pos_{u_j}	Position of a query/update in the query/update queue
t_{q_i}/t_{u_j}	Arrival time of a query q_i / an update u_j, the time of inclusion in the respective queue
rt_{q_i}	Retention time of a query q_i
P_{q_i}/P_{u_j}	Set of partitions processed by a query q_i / an update u_j
$uu(p)$	Number of unapplied updates on a partition p
qos_{q_i}	QoS value of a query q_i, $qos_{q_i} \in [0,1]$
qod_{q_i}	QoD value of a query q_i, $qod_{q_i} \in [0,1]$
$w(u_j)$	Weight of an update u_j

appropriate scaling of the system but not with the help of the approach proposed here.

Each query or update is associated with a timestamp t_{q_i} or t_{u_j}, respectively, which reflects the arrival time of the query or update in the system. The position of a transaction in the appropriate queue is denoted with pos_{q_i} for queries and pos_{u_j} for updates. The value 0 for both position parameters specifies the head of the queue, i.e., the transaction at this position is executed in the next run if the appropriate queue is selected. Additionally, each query q is enriched by a pair $\langle qos_{q_i}, qod_{q_i}\rangle$, where $qos_{q_i} \in [0,1]$ and $qos_{q_i} + qod_{q_i} = 1$. A higher value for qos_{q_i} denotes a higher demand for fast response times by the appro priate user, whereas a higher value for qod_{q_i} signifies a higher freshness demand. The time between the arrival and the execution of a query is called retention time rt_{q_i}, and it is used to measure the overall response time (QoS).

The scheduling of queries and updates in the context of data warehouses is not necessarily suited for every type of query. In this paper, we only want to focus on ad hoc queries that can be parameterized individually by the user who issued the query. However, on the analysis side, there are more query types, such as batch query workloads and user sessions. Regarding batch query workloads, the following tasks are usually addressed: the building or updating of materialized views, the refreshing of indexes, and the execution of batch queries with the purpose to build complex reports. These individual operations are not independent regarding the time dimension and, given the provided consistency requirements, there is absolutely no flexibility to switch certain operations. Aside from that, the parameterization in terms of QoS and QoD values does not make a lot of sense for batch workloads because these are executed at times defined by the system administrator.

User sessions represent a different approach; here, multiple queries are grouped to form one transaction. Typically, the user begins a session with a very raw view on the data and then approaches the relevant data details via drill downs or via the addition of predicates. Modifications on the base data within a session have to be handled by special routines in order to avoid that the user sees inconsistent data. Typical approaches here tackle this challenge with the help of multi version concurrency control [28,29]. It might be possible though that the user specifies the desired consistency requirements before the start of a session; as a consequence, this would allow updates within a session. In other words, a user is willing to take the risk to be confronted with inconsistent data within a session but in exchange, the same user receives more up to date query results. The QoS or QoD values can then be specified either globally for the full session or in a query related way within the session.

3.2. Partitioning

To measure the freshness of a query q, the updates that correlate with this query have to be identified. The simplest assumption would be to treat the whole data warehouse as one large data item, i.e., each update correlates with each query; however, this leads to unfeasible system premises, since a prioritization of updates is not possible under these circumstances. To improve the flexibility regarding the update prioritization (see Section 4.2.2), we subdivide the data warehouse (DWH) into a set of disjoint partitions P, $DWH = \{p_i | 1 \leqslant i \leqslant n\}$, where n is the number of partitions. There fore, we choose some dimensional elements of the multi dimensional data model to define the granularity of the partitions. A closer look at different partitioning models and their impact on the scheduling quality can be found in Section 5. The bottom line at this point is that we are able to control the accuracy of query/update correlations by applying different partitioning schemes. A fine partition ing scheme allows a more precise mapping between queries and updates with the trade off that a higher number of partitions need to be maintained.

Each user query q and each update u reads or writes, respectively, one or more partitions ($|P_q| \geqslant 1$ and $|P_u| \geqslant 1$). The number of unapplied updates in a partition p with respect to the available updates in the update queue is denoted as $uu(p)$ and is used to measure the freshness (QoD), as illustrated in the next section.

3.3. Performance measures

In this section, we describe our QoS and QoD metrics to measure the quality of our scheduling algorithm operating on a workload W (a set of queries and updates within a given time period).

3.3.1. QoS metric

The QoS criterion for a given workload W is defined by the *average retention time*, i.e., the average time between the arrival (t_{q_i}) and the execution (et_{q_i}) of all queries $q_i \in W$ processed by the system:

$$QoS(W) = \frac{\sum_{q_i \in W_q} et_{q_i} \quad t_{q_i}}{|W_q|}.$$

Hence, the retention time denotes the time the query had to wait for system resources and does not take into account the execution time.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

3.3.2. QoD metric

In order to measure the quality of data, we use a lag based approach, i.e., we take the number of unapplied updates to measure the staleness of a query result. More precisely, the QoD metric of a query q_i is affected by an update u_j if both access the same partition ($|P_{q_i} \cap P_{u_j}| = 1$) and if the arrival of the update lies before the arrival of the query: $t_{u_j} < t_{q_i}$. The second condition assures that staleness is computed only from those unapplied updates that are already in the system at the time the user runs the query; this is because the user does not expect to see "future results." To compute the overall QoD metric of a query, we select the most out dated partition $p_i \in P_q$:

$$QoD(q) = \min_{p_i \in P_q} \left(\frac{1}{1 + uu(p_i)} \right).$$

We normalized the QoD metric, so that $QoD(q)$ of a query q is 1 if there are no pending updates and it converges to 0 with an increasing number of unapplied updates. Due to the straightforward aggregation seman tics of the lag based approach, we decided against other schemes such as time or divergence based approaches. The QoD metric for a complete workload is then computed as follows:

$$QoD(W) = \frac{1}{|W_q|} \sum_{q_i \in W_q} QoD(q_i).$$

3.4. Optimization goal

Given a workload W with concurrent queries and updates, where each query $q_i \in W$ is enriched by a vote pair $\langle qos_i, qod_i \rangle$, the overall optimization goal is to minimize the retention time (QoS metric) for each query $q \in W_{qos}$, where $W_{qos} = \{q \in W_q | qos_q \geqslant 0.5\}$, and to max imize the freshness (QoD metric) for each query $q \in W_{qod}$, where $W_{qos} = \{q \in W_q | qod_q > 0.5\}$. Both metrics, QoS and QoD, cannot be compared with each other, i.e., a specific in /decrease of unapplied updates for a query q_i is not comparable to a specific in /decrease of the retention time of the same query q_i. Therefore, users have to decide for one side, i.e., they have to opt for a specific optimization criterion and they may do so with varying intensity. Considering the total weight of all votes, the system tries to find a compromise to meet the user requirements as closely as possible.

4. Scheduling

Since the requirements defined by the data warehouse users change over a certain period of time and since different users have different or even conflicting demands, we need a scheduling algorithm that dynamically adjusts itself to the workload changes and that ensures a fair balancing between the different parties of user groups (to avoid starvation). Traditional hard coded scheduling schemes such as FIFO, FIFO QH and FIFO UH cannot achieve this goal, since they do not adapt to workload changes and either prefer queries or updates, i.e., they support QoS or QoD user requirements. Therefore, we propose WINE, a two level scheduling scheme with two separated queues for queries and updates. At a higher level, we allocate resources by majority vote to either the query queue or the update queue (QoS versus QoD). At a lower level, we pick queries to prioritize according to their QoS values and we prioritize updates according to the QoD values of the affected queries.

4.1. First level scheduling

The scheduling algorithm on the whole is sketched in Algorithm 1. First level scheduling is straightforward: before a transaction, query or update is executed, the total sum of all the queries' QoS and QoD values in the queue is calculated (see line 18). If a query is executed or a new query arrives at the queue, both $\sum QoS$ and $\sum QoD$ can be incrementally maintained. For an example, see Fig. 3, where $\sum QoS = 3.7$ and $\sum QoD = 2.3$; hence, a query is executed in the next step. So, by holding the simple majority, the dominating user group is able to determine the appropriate system mode: the *query* or the *update mode*. Independent of the first level scheduling, we introduce a second level scheduling, which additionally supports the enforcement of the user preferences. This second level scheduling is done for queries if $\sum QoS \geqslant \sum QoD$ and for updates if $\sum QoS < \sum QoD$ (see lines 9 14).

4.2. Second level scheduling

In the presence of user requirements, WINE dynami cally allocates system resources to one of the transaction queues under consideration of the sums of the QoS or QoD values, respectively, of all queries. The user require ments, however, can be instrumentalized with much more detail, e.g., by prioritizing queries with large QoS values or by inserting updates much earlier when their associated queries dispose of large QoD values. In the following, we will show two techniques to prioritize single queries and updates based on the respective user demands.

4.2.1. Query prioritization

As a first optimization step, we try to minimize the retention time for queries with high freshness demands, i.e., queries q_i with $qos_{q_i} \geqslant 0.5$ (see Algorithm 1, line 26). Therefore, the queries $q_i \in Q$ are sorted in descending order of their QoS values qos'_{q_i} (a copy of the original qos_{q_i} values) and in ascending order of their timestamps t_{q_i} (line 31). Thus, queries with a high QoS value are favored by the system, whereas queries with a low QoS value (and hence with a high QoD value) are delayed in their execution; this allows to apply corresponding updates in the meantime, i.e., to reduce the staleness of such queries. Obviously, this can easily lead to the starvation of queries with low QoS values if queries with higher QoS values keep arriving. Hence, with an increasing retention time, the rank of a query needs to be raised. Therefore, we increment the QoS value of all elements in the query queue after each execution of a query (line 29). Over time, this gives queries with lower QoS values an advantage

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

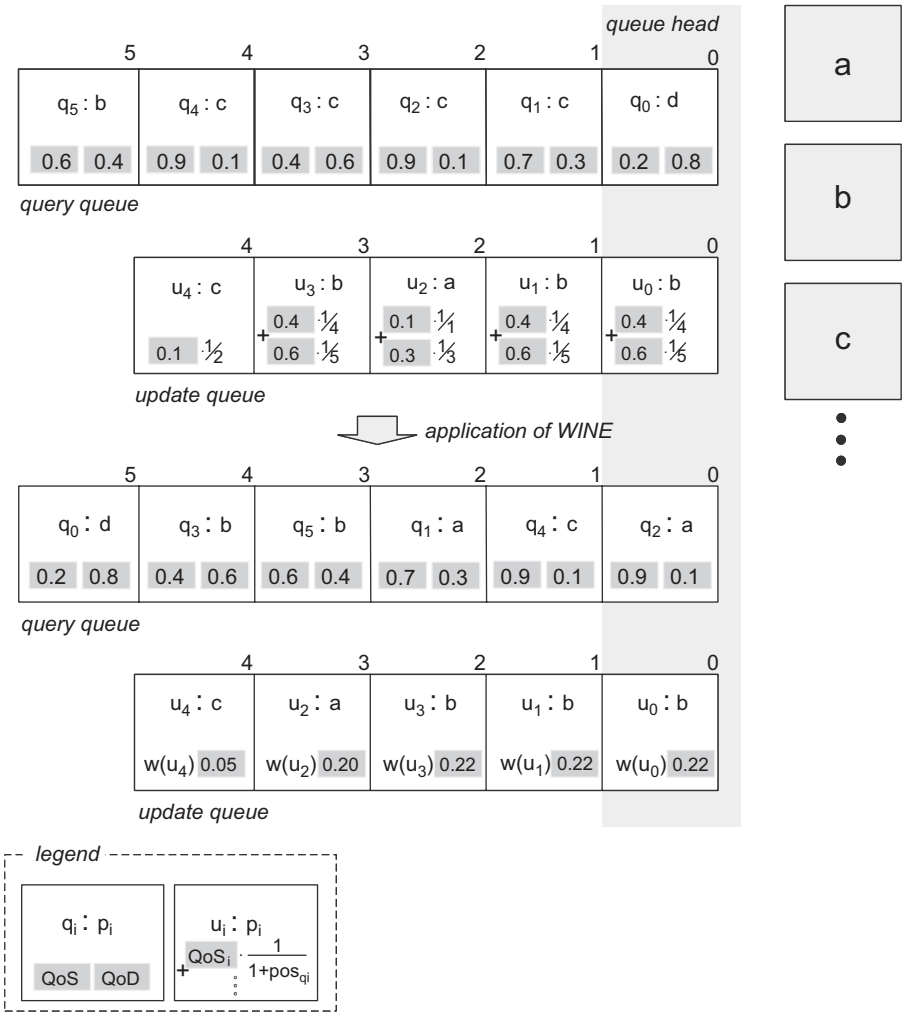

Fig. 3. WINE example.

over queries with later arrival times. The delta d by which the QoS values are adjusted depends on the query queue length $|Q|$ (with the length being re determined with every respective query prioritization) and is computed as follows:

$$d = \frac{1}{|Q| \cdot r_{max}}.$$

It is the reciprocal of the query queue length $|Q|$ multiplied by a system parameter r_{max}, which defines the maximum retention time of a query in the system. A value 1 for r_{max} means that a query needs at most $|Q|$ 1 query executions to be scheduled for execution. For $r_{max} = 5$, it needs at most $(|Q|$ 1$) \cdot 5$ executions, and so on. A significance test on r_{max} is given in Section 6.4. The original QoS value qos_{q_i} of a query q_i is not changed by the operation on line 29, and with it, the first level scheduling remains untouched.

Algorithm 1. WINE scheduling algorithm

1: Q: query transaction queue
2: U: update transaction queue
3: *mode*: specifies the system mode
4: r_{max}: maximum query retention time
5:
6: WINE:
7: **while** $|W| > 0$ **do**
8: **execute** GLOBALSCHEDULING
9: **if** *mode* = *querymode* and $|Q| > 0$ **then**
10: **execute** QUERYPRIORITIZATION
11: **end if**
12: **if** *mode* = *updatemode* and $|U| > 0$ **then**
13: **execute** UPDATEPRIORITIZATION
14: **end if**
15: **end while**
16:
17: GLOBALSCHEDULING:
18: **if** $\sum QoS \geqslant \sum QoD$ and $|Q| > 0$ **then**
19 **execute** query q **where** $pos_q = 0$
20 *mode* *querymode*
21: **else**
22: **execute** update u **where** $pos_u = 0$
23: *mode* $\leftarrow$ *updatemode*
24: **end if**

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

```
25:
26:  QUERYPRIORITIZATION:
27:  d    (|Q| · r_max)^-1
28:  for all q ∈ Q do
29:     qos'_q    min(qos'_q + d, 1)
30:  end for
31:  sort Q by qos'_q, t_q
32:
33:  UPDATEPRIORITIZATION:
34:  for all u ∈ U do
35:     w_u ← 0
36:     for all q where P_q = P_u do
37:        w_u ← w_u + qos_q · 1/(1+pos_q)
38:     end for
39:  end for
40:  sort U by w_u, t_u
```

As we saw earlier, query prioritization has two benefits: (1) it decreases the retention time of queries with increasing QoS values, and (2) it delays the execution of queries with decreasing QoS values, i.e., the probability that upcoming updates for these queries are applied increases (directly supports their QoD demands).

Starvation freedom: As we illustrated earlier, an increase in the time a query q exists in the system leads us to increment the QoS value qos_q of this query by d after every query execution. Since d continuously decreases with growing queue length, we only need to show at this point that the scheduling of queries is also starvation free if new queries with high QoS values continue to trickle into the system. The notion of starvation free means that the system guarantees that all ready jobs will eventually run, regardless of the workload and under the assumption that the arrival rate of new jobs is smaller than or equal to the maximal throughput of the system. That is to say, we want to know whether or not a query q_i with a low QoS value can prevail when faced with a continuous stream of queries with QoS values of 1. In this case, d slowly approximates 0 but the sum of the increments added to the QoS values of the queries never converges, i.e.:

$$\lim_{n\to\infty} \sum_{|Q|\ 0}^{n} \frac{1}{|Q| \cdot r_{max}} = \infty.$$

That is to say, a query q_i with a low QoS value will never starve, even if new queries with QoS values of 1 are continuously added to the queue. The reason is that the QoS value of q_i will reach a value of 1 in any case. In case of identical QoS values, the timestamp t_{q_i} is used as prioritization criterion, which means query q_i will be preferred, since it has been in the system for a longer time period compared to newly arriving queries. In other words, for all queries with a QoS value of 1, the query scheduling is based on the FIFO principle. Hence, the scheduling algorithm for queries is starvation free.

4.2.2. Update prioritization

In addition to the latter improvement, we want to prioritize updates for which the corresponding queries will be executed soon, i.e., queries which are closer to the query queue head. Therefore, we introduce a weight $w(u)$ for an update u, which is computed as follows:

$$w(u) = \sum_{\forall q_i,\ P_{q_i} \cap P_u \neq \emptyset} \frac{qod_{q_i}}{1 + pos_{q_i}}.$$

The QoD value of each query that requests the same partition as the update ($|P_{q_i} \cap P_u| \neq \emptyset$) is weighted by its position in the query queue. The sum of all those weighted QoD values gives the overall weight $w(u)$ (see lines 34 39). The sooner a query is executed, the higher the weights of the corresponding update. The update queue is sorted in decreasing order of the resulting weights w_u and in increasing order of the timestamps t_u (see line 40). The second order criterion is very important in terms of consistency as it avoids that a sequence of updates that refer to the same partition get interchanged. Two updates u_i and u_j that refer to the same partition p and are thus in a potential conflict with each other always have the same weight ($w(u_i) = w(u_j)$); hence, the timestamps t_{u_i} and t_{u_j} preserve the original update order.

In contrast to query scheduling, the scheduling of updates is not starvation free. That is to say, workloads may be constructed that will never allow the execution of a specific update. However, since update scheduling takes the user demand into consideration, this does not represent a disadvantage but a desired feature of the system. In detail, the following differentiation options arise:

(1) If there are no queries q_i for an update u_j that meet the condition $P_{q_i} \cap P_{u_j} \neq \emptyset$, this results in a weight of $w(u_j) = 0$ (line 35 in Algorithm 1). That is to say, update u_j is only executed if no other updates with a weight larger than 0 exist in the system at the given point in time. Update u_j will be put on hold until a correlated query is sent to the data warehouse.
(2) If there is at least one query q_i for an update u_j that meets the condition $P_{q_i} \cap P_{u_j} \neq \emptyset$, this means that u_j will be scheduled in dependence on all other updates and their correlated queries. Since queries never starve, as shown in Section 4.2.1, the value pos_{q_i} of the correlated query q_i slowly converges to 0, which results in the continuous increase in the weight of update u_j and hence, the execution likelihood of u_j also increases.

4.2.3. Example

We will sketch the query and update prioritization with an example. In Fig. 3, we see the initial state of the system with six queries and five updates in the queues and three partitions A, B and C. For illustration purposes, each query and each update refer to exactly one partition. The QoS and QoD values of the queries are highlighted by the gray boxes. The order of the query queue after query prioritization (ordered by qos_q) is as follows: q_0, q_3, q_5, q_1, q_4 and q_2, with q_2 at position 0. This new query schedule is utilized by the update prioritization to compute the weight of each update. The computation of the weights is sketched by the gray box in the update elements. We pick update u_2 to illustrate this step: we have query q_2 at

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

position 0 and query q_1 at position 2; they both refer to partition A and have QoD values of 0.1 and 0.3, respectively. Now, the weight $w(u_2)$ is computed as follows:

$$w(u_2) = qod_{q_2} \cdot \frac{1}{pos_{q_2}} + qod_{q_1} \cdot \frac{1}{pos_{q_1}}$$
$$= 0.1 \cdot \frac{1}{1} + 0.3 \cdot \frac{1}{3} = 0.2.$$

The new order of the update queue after the update prioritization is u_4, u_2, u_3, u_1 and u_0, with u_0 at position 0. So, u_2 is ranked at the 4th position, since the QoD values of the correlating queries q_1 and q_2 are comparatively low, i.e., the freshness demands of the users having initiated these queries are not very high.

5. Partitioning model

In Section 3.2, we stressed that the partitioning of data is necessary in order to detect correlations between queries and updates and to exploit these information for the scheduling of updates (second level scheduling). For this purpose, we introduced a very abstract model but we did not go into details in terms of the criteria relevant for the data partitioning, and neither did we consider how the partitioning affects the scheduling and how partitions are addressed by queries and updates. These items shall be analyzed in more detail in the following section.

5.1. Motivation

Before we take a closer look at the partitioning model, we want to illustrate with an example how the selection of a certain partitioning schema affects the scheduling. For this, we take the base relation from Fig. 4, which consists of two dimension attributes A and B and a measure attribute M. There are four possibilities to aggregate this relation: (1) aggregation without considering the dimen sion attributes, (2) aggregation based on attribute A, (3) aggregation based on attribute B, or (4) aggregation based on A and B (see Fig. 4 from top to bottom). Now, we would like to show how the scheduling process is affected by the selection of the partitioning schema. To do so, let us have a look at the workload from Fig. 5, which consists of two queries and two updates, which address partitions a and b or c and d, respectively. Now, in order to measure the influence of the different partitions on the scheduling, we have to take a look at the weightings $w(u_0)$ and $w(u_1)$, which result from the different partitions.

Partition schema		$w(u_0)$		$w(u_1)$	
P_1	()	$0.8 \cdot \frac{1}{2} + 0.1 \cdot \frac{1}{1}$	0.5	$0.8 \cdot \frac{1}{2} + 0.1 \cdot \frac{1}{1}$	0.5
P_2	(A)	$0.1 \cdot \frac{1}{1}$	0.1	$0.8 \cdot \frac{1}{2}$	0.4
P_3	(B)	$0.8 \cdot \frac{1}{2} + 0.1 \cdot \frac{1}{1}$	0.5	$0.8 \cdot \frac{1}{2} + 0.1 \cdot \frac{1}{1}$	0.5
P_4	(A, B)	$0.1 \cdot \frac{1}{1}$	0.1	$0.8 \cdot \frac{1}{2}$	0.4

As we can see, the weightings differ depending on the partition. For $P_1 = ()$ and $P_3 = (B)$ (also refer to Fig. 4(1) and (3)), the weightings are identical for u_0 and u_1, respectively, since the exact assignment of updates to the appropriate queries is impossible due to wrong partitioning, and thus, the QoD values of the queries are equally distributed to both updates. This has the effect though that the updates cannot be prioritized by using their weightings and that they can only be scheduled based on their arrival times. When using partitionings $P_2 = (A)$ and $P_4 = (A, B)$, however (also refer to Fig. 4(2) and (4)), update u_0 can be assigned to query q_0 and update u_1 can be assigned to query q_1, which means that a precise calculation of the weightings becomes possible. Even though both partitionings P_2 and P_4 return the correct result, we would decide for partitioning schema P_2 for the workload at hand because it takes up less space, i.e., it comprises only 2 instead of 4 tuples. From this example, it becomes clear that the selection of the partitioning schema affects the quality of the scheduling for updates significantly. Furthermore, we see that different partitioning schemas may return the same result but differ in the size of the resulting data structure. The size, however, directly affects the look up costs, i.e., the duration of the calculation of correlations between queries and updates; these should be kept as low as possible. The semi automatic determination of the partitioning schema for a given workload under consideration of these two criteria will be analyzed in detail in the following section.

5.2. Workload aware partitioning

As our example in the previous section illustrated, the partitioning schema required for the realization of

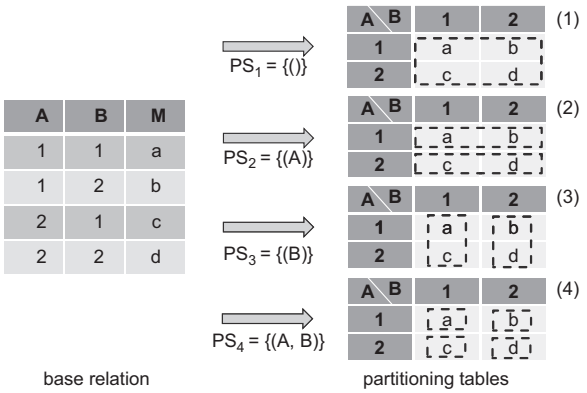

Fig. 4. Partitioning example.

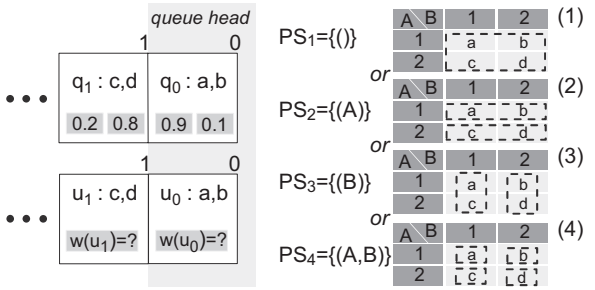

Fig. 5. Example workload for different partitioning schemas.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

scheduling must not be defined arbitrarily or independent from the workload, since this would undermine the prioritization of updates. Therefore, we will use the next section to take a look at the generation of attribute candidates that will be decisive for the partitioning. First, we will explain the term *partitioning schema*.

Partitioning schema: A partitioning schema *PS* consists of a set of attribute sets that constrain the granularity and range of data within the multi dimensional data structure; $PS := \{AS_1, \ldots, AS_n\}$. Each attribute is prefixed with the dimension identifier.

A valid partitioning schema for the TPC H database, for example, would be $\{(Store.Nationkey, Customer.Nationkey)\}$.

In order to support the scheduling of queries and updates to the best possible extent, we need to identify those predicates and attributes that appear rather frequently and often together in the queries and updates. In order to do so, attribute sets AS_i appearing in the WHERE clauses of all queries and updates are considered and annotated with their frequency of appearance, resulting in pairs, (as_i, o_i), each containing an attribute set and its occurrence frequency; these will be stored in two separate lists. The resulting lists, L_q for the query attribute sets and L_u for the update attribute sets, are then merged to compose the list of partitioning attribute sets, L_p. The individual steps involved here are shown in Algorithm 2. We scan through both attribute lists, L_q and L_u, and insert those value pairs into the result list L_p whose attributes appear in both lists and whose occurrence frequency in at least one of the lists is larger than the threshold o_{min}. For the occurrence frequency, we decide for the maximum from the two occurrences. Additionally, the empty set is also an element of L_p.

As can be seen easily, the selection of the system parameter o_{min} strongly depends on the workload. If the workload consists of queries and updates with many attribute sets of a low occurrence frequency, o_{min} has to be set to an appropriately low value in order to receive sufficient coverage in the attribute sets of the result list L_p. However, we believe that in practical scenarios, the frequency of an attribute is not distributed equally but there always exist a few attribute sets that are of strong interest and that are subsequently addressed more often in the workload. In this case, parameter o_{min} has to be set to a value that guarantees that precisely those frequent attribute sets will be the result of Algorithm 2.

Algorithm 2. Generation of partitioning attribute candidates

Require L_q: list of query attributes
Require L_u: list of update attributes
Require o_{min}: minimum ratio
Require $L_p \leftarrow \emptyset$: result list of partitioning attributes

```
forall q ∈ L_q
  forall u ∈ L_u do
    if q.as = u.as ∧ (q.o > o_min ∨ u.o > o_min) then
      L_p ← L_p ∪ (q.as, MAX(q.o, u.o))
    end if
  end for
end for
L_p  L_p ∪ ∅
```

The attribute sets AS_i from the result list L_p form a lattice $Ł = (N, E)$, with the node set being defined as the power set of all attribute sets of the result sets L_p, i.e., $N = P(L_p)$, and with the edge set being defined as $E \subseteq N \times N$ with $(x, y) \in E$ if and only if $x \subset y$, but there is no element between x and y, i.e., $\neg\exists z$ with $x \subset z \wedge z \subset y$. An example for such a lattice Ł, derived from the result list L_p with the minimum occurrence frequency $o_{min} = 5\%$, is illustrated in Fig. 6. Aside from the attribute sets, Fig. 6 also shows the sum of the occurrence frequency accumulated from the root to the leaves. Since the result list L_p only contains the attribute sets whose occurrence frequency is larger than the threshold o_{min}, the accumulated sum of the occurrence frequencies is strictly monotonic increasing. The cumulated sum of the occurrence frequencies of the attribute sets (A, B), for example, is calculated from the occurrence frequencies of the attribute sets (A), (B) and (A, B) itself $(o_A + o_B + o_{AB} = 10\% + 10\% + 5\%)$. Attribute sets such as (D, E), for example, whose occurrence frequency is too small (here: 2%), are not part of the tree and thus shaded in gray in Fig. 6. In order to keep the number of attribute sets in the partitioning schema as small as possible, those attribute sets that allow to derive all predecessor attribute sets are to be selected at the leaves of the tree. In our example, these are the underlined attribute sets (A, B, C), (D) and (F, G, H). In the worst case scenario, all attribute sets are disjoint, i.e., no attribute set can be derived from another one (except for the empty set), and thus the number of attribute sets in the resulting partitioning schema is $|L_p|$ 1. Due to the explorative character found in multi dimensional data analyses, however, this case rarely occurs in practical settings. Often starting from a raw view on the data, the user will drilldown to navigate to the actual data object of interest, e.g., from (A) to (A, B) to (A, B, C) in Fig. 6.

For every attribute set found with the above procedure, a materialized view is created, which we will denote as partitioning table PT_i. The rule for the creation of the partitioning table is a SELECT clause with the attributes of the attribute sets in the GROUP BY clause. For the attribute set from the example above, (A, B, C), this results in the partitioning table $PT_{(A,B,C)}$.

Example: We want to look at an example from a TPC H database with the scaling factor 0.1, i.e., the *lineitem* table consists of 600,000 tuples. The cardinality of the partitioning attributes *p_brand*, *s_nationkey*, *c_nationkey* is

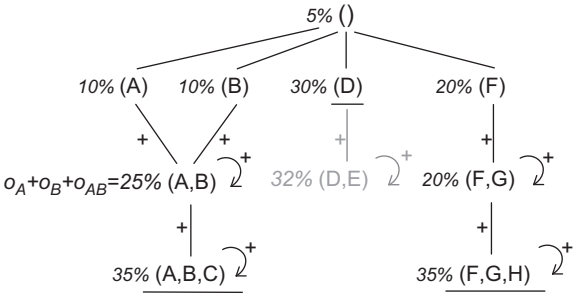

Fig. 6. Attribute set lattice Ł with o_{min} 5%.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

25 each. That is to say, there are 25 different brands in the *parts* table and 25 different nations in the *supplier* and *customer* tables, respectively. For the partitioning schema

$PS_1 := \{(p\_brand, s\_nationkey, c\_nationkey)\}$,

this results in a partitioning table PT_1 with 15,625 tuples ($= 25^3$), i.e., an average aggregation ratio of 1:39. For a less detailed partitioning schema

$PS_2 := \{(s\_nationkey, c\_nationkey)\}$,

the result is a partitioning table PT_2 with 625 tuples ($= 25^2$), i.e., an average aggregation ratio of 1:960.

5.3. Correlation between queries and updates

In order to create a correlation between a query q_i and an update u_j, these are re written in such a way that they only reference the attributes of the partitioning schema and then they are executed in this form on the partition ing table PT_i. If the intersection of the result sets $R_{q_i} \cap R_{u_j}$ is not empty, q_i and u_j address the same data and a correlation has been found. However, this correlation implicitly comes with a certain probability, since the partitioning schema only roughly categorizes the data. A correlation might be detected with probability 1 if the partitioning attributes were to consist of the key of the fact table. In return, this would mean, though, that the partitioning table has the same cardinality as the fact table, i.e., the execution of queries or updates on the partitioning table would be identical to their actual execution. The effects of the granularity of the partitioning schema on the accuracy of the matching will be considered in the next section.

5.4. Selectivity based exception handling

As already explained in the previous section, the partitions are created from the subset of all attributes of the given workload. This in turn means that the resulting partitions aggregate and represent several hundreds or thousands of tuples. Since queries or updates may contain further attributes in their WHERE clauses in addition to the attributes of the partitioning schema, matching partitions from a query q and an update u ($P_q \cap P_u \neq \emptyset$) do not necessarily guarantee that the result set of q matches the updated set of u. This leads to the following problems: (1) If an appropriate update u_a has been found for query q_a via the comparison of partitions, this query will spend unnecessary time waiting for the insertion of u_a; the reason is that no overlappings can be found within the partition when taking a closer look at all predicates. (2) Update u_a is scheduled based on the QoD value of query q_a even though no matchings are detected when analyzing the result set and the update set in detail, i.e., updates are scheduled on wrong assumptions.

We can calculate the probability of overlappings between the result set of a query q with selectivity s_q and the update set of update u with selectivity s_u. Since the selectivity estimation takes place on the level of partitions, the selectivity is only estimated from the residual predicate attribute set R, which does not belong to the attributes of the partitioning schema PS, i.e., $PS \cap R = \emptyset$. In order to estimate for a partition with cardinality N, we use the hypergeometric distribution. For this, a random sample of size n is drawn from a basic population without replacement. The hypergeometric distribution gives the probability for the event that a certain number of elements with the desired attributes appear in the sample. In general, if a random variable X follows the hypergeometric distribution with parameters N, M and n, the probability of getting exactly k successes is given by

$$h(k; N, M, n) = \binom{M}{k}\binom{N-M}{n-k} \Big/ \binom{N}{n}.$$

In order to simplify the problem, we will look at the opposite case, in which no attribute matchings are found, i.e., $k = 0$. Parameter M is calculated from $M = s_q \cdot N$ and parameter n is found via $n = s_u \cdot N$. That is to say, the sample of size n is derived from all the tuples referenced by update u with selectivity s_u in the complete partition of size N. Then, the tuples of query q, $s_q \cdot N$, are checked against this sample. Now, we can use these formulas to calculate the probability $p(N, s_q, s_u)$ for the event that a query with selectivity s_q and an update with selectivity s_u access at least one shared tuple in a partition of size N:

$$p(N, s_q, s_u) = \begin{cases} 1 - \binom{N - s_q \cdot N}{s_u \cdot N} \Big/ \binom{N}{s_u \cdot N}, & s_q + s_u < 1 \\ 1, & \text{otherwise.} \end{cases}$$

In case the sum of the selectivities exceeds the value 1, there is always a shared tuple, i.e., $p(N, s_q, s_u) = 1$.

After having derived the analytical formula that calculates the probability of a matching for a query update pair, we now want to take a closer look at the parameters N, s_q and s_u and examine their impact on the matching probability $p(N, s_q, s_u)$. In order to do so, we assume an average update selectivity s_u of 10%, i.e., 10% of the tuples in a partition are referenced by an update. Fig. 7a illustrates the matching probability for a growing query selectivity s_q and different partition cardinalities N. As we can see, the matching probability for larger partitions already reaches the value 1 for low query selectivities of 1% and less. Fig. 7b shows this in more detail for high selectivities between 1% and 10% and growing partition sizes. As illustrated, the matching probability for decreasing partitions increases at a slower pace and it only reaches the value 1 for selectivities close to 1. However, one might argue that the partitioning schema of small partitions i.e., of finer granularities already contains many attributes of the workload and thus, the selectivity specified by the small number of additional selection attributes tends to be low. Fig. 7c shows the matching probability for varying update and query selectivities. As can be seen, the matching prob ability for high update selectivities s_u between 0.5% and

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

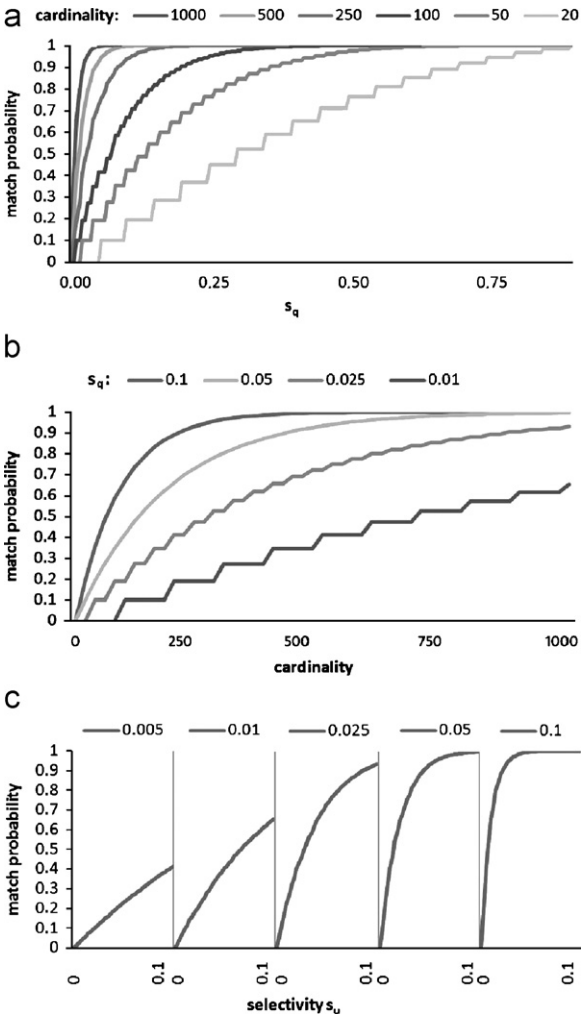

Fig. 7. Matching probabilities for different system parameters: (a) increasing query selectivity s_q, update selectivity s_u 0.1 and different partition cardinalities N, (b) increasing cardinality N, update selectivity s_u 0.1 and different query selectivities s_q, (c) partition cardinality of 1000, different query selectivities s_q and an increasing update selectivity s_u between 0% and 10%.

1% is only significant for accordingly low query selectiv ities close to 10%.

From the analysis of the matching probability for all query update pairs of a workload history, we can derive two conclusions: (1) Given that the matching probability remains equally high for all query update pairs and assuming that the workload will not change significantly in the future, the previous matching criterion $P_{q_i} \cap P_u \neq \emptyset$ is sufficient. (2) For strong fluctuations in the matching probability within a workload history or in cases when the exact composition of a future workload is not known, the matching criterion has to be extended to $P_{q_i} \cap P_u \neq \emptyset \wedge p(N, s_{q_i}, s_u) > MPT$. Here, MPT denotes the matching probability threshold, and its value beyond which a match is assumed has to be specified by the system adminis trator. The calculation of the weightings for the updates from 4.2.2 then has to be extended as follows:

$$w(u) = \sum_{\forall q_i,\ P_{q_i} \cap P_u \neq \emptyset \wedge p(N, s_{q_i}, s_u) > MPT} \frac{qod_{q_i}}{1 + pos_{q_i}}.$$

That means we introduce an exception handling method for those updates, where either their own selectivity and/or that of their correlated queries is too high.

6. Experiments

We conducted an experimental study to evaluate the stability and performance of WINE with respect to the other scheduling algorithms mentioned in Section 1. Furthermore, we measured the adaptivity of WINE with respect to different trends of user requirements in the workload. In summary, we found that WINE has the following properties:

- It reacts rapidly to changing user preferences.
- It outperforms all competing algorithms in the entire spectrum of user requirements and workloads. In fact, WINE performs almost as well as FIFO QH with respect to QoS and as well as FIFO UH with respect to QoD. Both baseline algorithms are the best in their dis ciplines.

6.1. Experimental setup

Our experimental setup consists of a middleware to implement the scheduling and balancing and a database server to store the data warehouse. Both are located on different machines: an Intel Pentium D 3.0 GHz system running Windows XP with 2 GB of main memory for the middleware and a dual CPU Xeon 64 Bit 2.8 GHz processor with 4 GB RAM running Linux and IBM DB2 V9.1 installed for the database server.

We implemented WINE as well as the FIFO, FIFO QH and FIFO UH scheduling schemes using Java 1.5. For the data warehouse, we made use of both synthetic and real world data. The synthetic datasets are based on the well known TPC DS database [30] and for our real world experiments, we used the CDBS database,[1] which contains information on radio and television broadcast services in the United States. The most important setup parameters for the experiments with the synthetic and the real world data, respectively, are illustrated in Fig. 8 and examples will be explained for the TPC DS setup in the following paragraph.

For the fact table, we took *web returns* with a scale factor of 1, i.e., 72,176 tuples. To implement the partition ing schema, we used the attribute *manufact_id* from the dimension table *item*, which is related to *web returns* through the foreign key *wr_item_sk*. The values of the attribute *manufact_id* lie in the range from 1 to 1000, which enables us to control the granularity in varying steps. The default granularity for our experiments is set to 50 partitions, with the smallest partition consisting of

[1] http://www.fcc.gov/mb/cdbs.html

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

		TPC DS (synthetic data)	CDBS (real world data)
schema	fact tbl (cardinality)	web returns (72,176)	application (585,433)
	dimension tbl	item	facility
	partitioning attribute	manufact_id	fac_zip1
part t ons	number	50	200
	smallest	101 tuples	12 tuples
	largest	2,572 tuples	15,432 tuples
transact ons	number of queries / updates	500 / 500	500 / 500
	query / update execution time	100 160 ms / 20 40 ms	630 720 ms / 420 840 ms
	load polling time LOW / MEDIUM / HIGH	100 sec / 75 sec / 50 sec	1000 sec / 750 sec / 600 sec

Fig. 8. Experimental setup.

101 tuples and the largest partition comprising 2572 tuples. To find correlations between transactions, the TPC DS queries and updates have been re written so that they address the partition attribute *manufact_id*.

To measure the behavior of WINE and the other scheduling algorithms under various workloads, we generated two sets of query and update traces, both consisting of 500 queries and 500 updates, which are polled in a time window of 100 and 1000 s, respectively: (1) random traces W_{random} with a mean of 10 queries and updates per second and a variance of 10, (2) traces with either 15 queries and 5 updates or 5 queries and 15 updates per second; alternations are due every 25 s and every 250 s, respectively ($W_{alternate}$). Query execution times range from 100 to 160 ms (630 720 ms for CDBS) and update execution times range from 20 40 ms (420 840 ms for CDBS). The workload polling time can be shrunk to simulate different degrees of load. As a matter of fact, we denote the original workload with a polling time of 100 s for the TPC DS dataset and 1000 s for the CDBS dataset as *LOW*. The polling times for the *MEDIUM* and *HIGH* workload for both datasets can be derived from Fig. 8.

The system parameter r_{max} is set to 5 for the following experiments. The impact of varying values for r_{max} will be analyzed in detail at the end, in Section 6.4.

6.2. Performance comparison

In the following section, we want to compare WINE and the three traditional scheduling algorithms FIFO, FIFO QH and FIFO UH for various given user preferences and workloads. FIFO, FIFO QH and FIFO UH are one level scheduling algorithms with different preferences: (1) FIFO does not prioritize any transaction; hence, transactions are executed in order of their arrival time. (2) FIFO UH always favors updates over queries. If the workload tends to consist of a lot of updates, FIFO UH is unable to process any queries. (3) FIFO QH always favors queries over updates. If queries keep arriving, FIFO QH does not execute any updates, i.e., the staleness increases.

This set of experiments has the following design: we measured the QoS and QoD metrics 30 times for both types of workloads (random and alternating) and varying values for QoS and QoD. Since the statistics of all these setup combinations show a very similar character, we aggregated them to the result outlined in the following.

We plot the QoS metrics (average retention time) for all four scheduling algorithms in Fig. 9a and b and differ entiate them into three levels of increasing load. We see that WINE and FIFO QH have a similar performance and outperform the other algorithms. For the TPC DS dataset, WINE performs, on average, 282.98% better than FIFO, 377.86% better than FIFO UH, and 15% worse than FIFO QH, which is self evident, since FIFO QH always favors queries over updates, i.e., FIFO QH represents the opti mum with regard to the QoS metrics. Due to the limited throughput of the system, the retention time for each scheduling schema increases with higher loads and the impact of the individual scheduling algorithms on the QoS metrics is continuously rising in significance. If the QoS metrics of the respective scheduling algorithms for the load level *LOW* are relatively close to each other (since there are no overloads of the system and the number of elements in the queues will always remain low), the advantage of WINE will become more visible with rising loads.

For the CDBS dataset, the situation is similar but less unique. Here, the QoS metrics for the individual schedul ing algorithms are closer to one another than for the TPC DS dataset. The reason is found in the fact that the execution times for the queries and updates were almost identical (in contrast to the TPC DS setup), which implied that the load could not be as high, since long updates represented an additional burden for the system.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

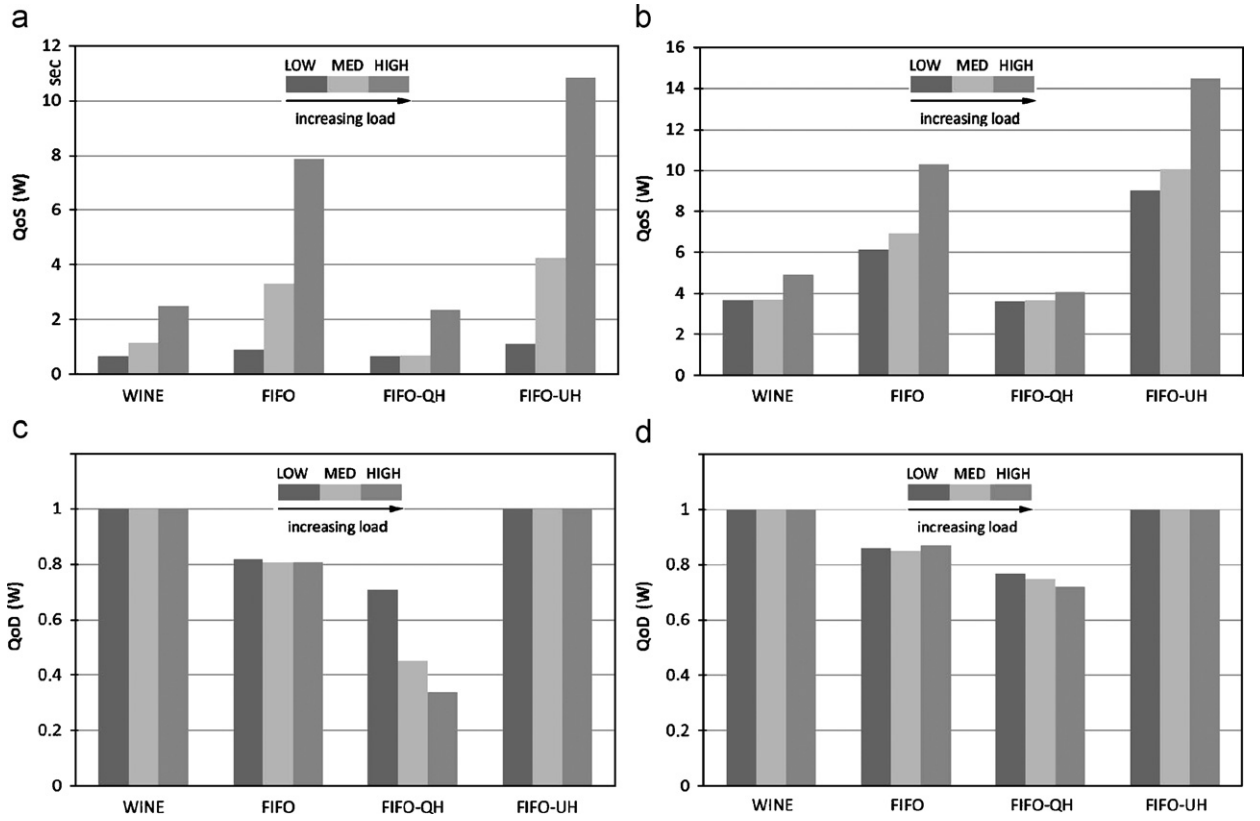

Fig. 9. System performance of WINE and the three baseline scheduling algorithms for synthetic and real-world datasets: (a) QoS performance for TPC-DS, (b) QoS performance for CDBS, (c) QoD performance for TPC-DS, (d) QoD performance for CDBS.

The direct comparison shows that the average retention time of WINE was 190.47% better than for FIFO, 274.19% better than for FIFO UH, and 7.49% worse than for FIFO QH.

Similar, Figs. 9c and d show the QoD metrics for all scheduling algorithms by varying the degree of load. We see that WINE and FIFO UH perform best with no unapplied updates at each load level: *LOW*, *MEDIUM* or *HIGH*. Note that FIFO UH represents the optimum with regard to the QoD metrics because it always favors queries over updates. For FIFO, the QoD metric remains stable with increasing load, since queries and updates are executed in order of their arrival time; so, on average, every other transaction is an update, i.e., the number of unapplied updates remains at a certain level. For FIFO QH, the QoD metric gets worse from the *LOW* to the *HIGH* load since an increasing load leads to more and more updates in the update queue that will not be applied because FIFO QH favors queries over updates. Here, too, it is apparent that the differences between the QoD metrics for the individual load levels are more significant for the TPC DS dataset than for the CDBS dataset. The reason also lies with the longer updates in the CDBS setup, since this is a considerable burden for the load and the difference becomes less significant.

Summarized, the main weakness of both FIFO QH and FIFO UH is their fixed preference for queries (QoS) or updates (QoD), which is a drawback in workloads with unpredictable user preferences.

6.3. Adaptability to user requirements

Having shown that WINE performs much better than the traditional hard coded scheduling algorithms in the general case, we want to illustrate that WINE quickly adapts to changing trends in user behavior for the TPC DS as well as the CDBS dataset. We use a random workload for each experiment in Figs. 10 and 11 and plot the arriving queries and updates. To illustrate the adaptability beha vior, we vary the $\langle qos, qod \rangle$ vote that comes with each query $q_i \in W_{random}$. More precisely, we divide the work load into four sections and invert the QoS and QoD values at each section transition. The QoS and QoD values in the first section are 0.1 and 0.9 in Figs. 10a and 11a and 0.4 and 0.6 in Figs. 10b and 11b. Thus, we have extreme turn arounds in trend in the first workload and slightly changing trends in the second workload.

Figs. 10 and 11 plot the QoS and QoD sums for all queries in the system during workload execution. Each vertical line marks a trend inversion, e.g., the QoS value of the queries changes from 0.1 to 0.9 and vice versa for the QoD value. We see that for the *LOW* workload, the QoS and QoD sums adapt to the changing trends very quickly, and

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

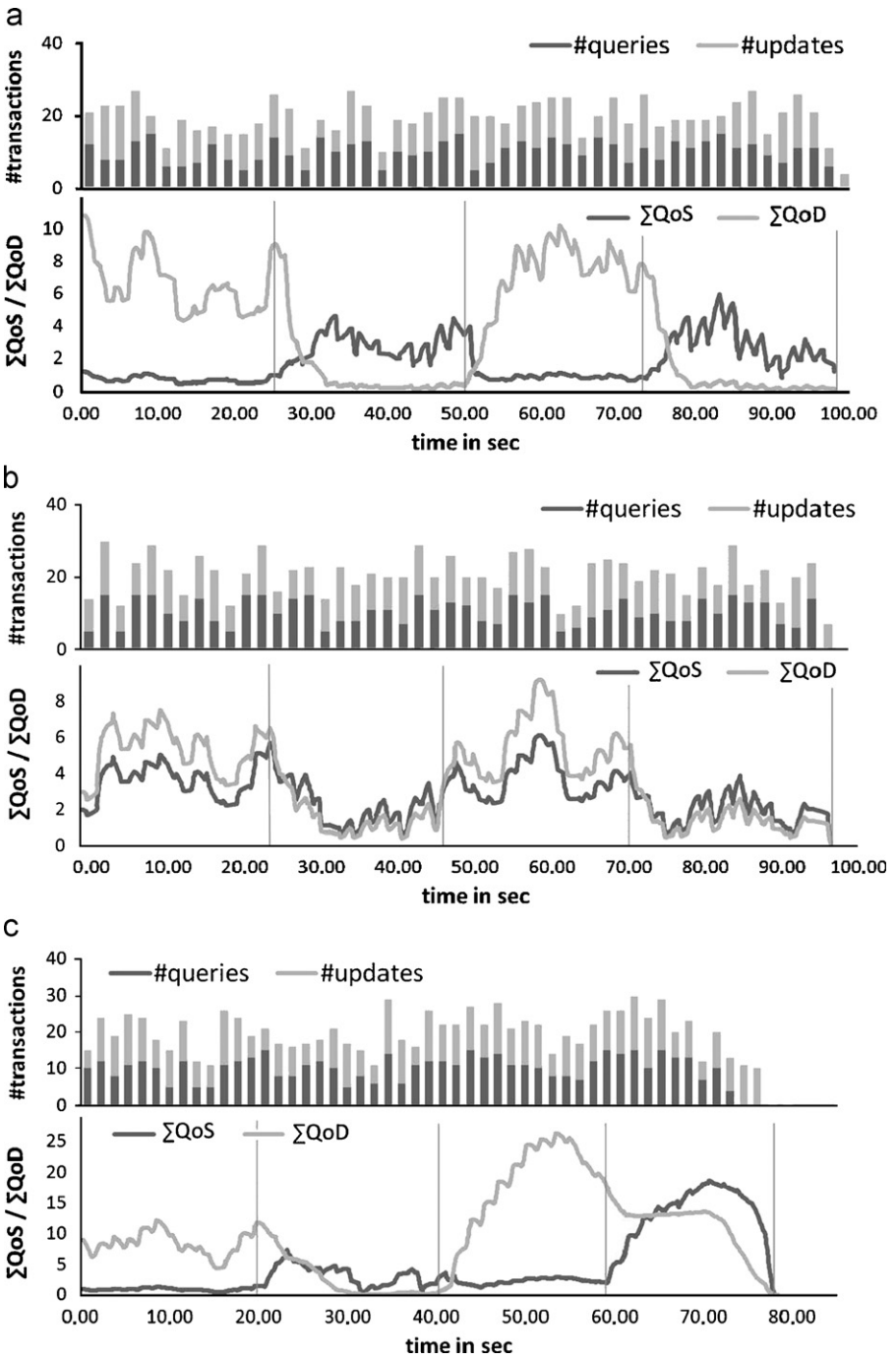

Fig. 10. Adaptability to user requirements for different workload settings on TPC-DS dataset: (a) *LOW* workload, $\langle QoS_1, QoD_1 \rangle \quad \langle 0.1, 0.9 \rangle$, (b) *LOW* workload, $\langle QoS_1, QoD_1 \rangle \quad \langle 0.4, 0.6 \rangle$, (c) *MEDIUM* workload, $\langle QoS_1, QoD_1 \rangle \quad \langle 0.1, 0.9 \rangle$.

with them, the overall system mode changes to *query mode* if $\sum QoS \geqslant \sum QoD$ and to *update mode* in the contrary case. A drop of the QoD graph in a QoD dominated section means that the update queue is empty and therefore queries are executed. The reason for the higher QoD graphs in all figures is that only a few queries are executed in QoD dominated sections and therefore, the QoD sum increases much more, since queries with high QoS value are executed first. With decreasing discrepancy between the specific user trends (Figs. 10b and 11b), the QoS and QoD graphs converge as expected, but the switch between the system modes during a trend inversion is accomplished as fast as before.

The two previous experiments on adaptability were performed with a *LOW* workload, i.e., a workload that does not lead to tailbacks in transaction processing. To

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

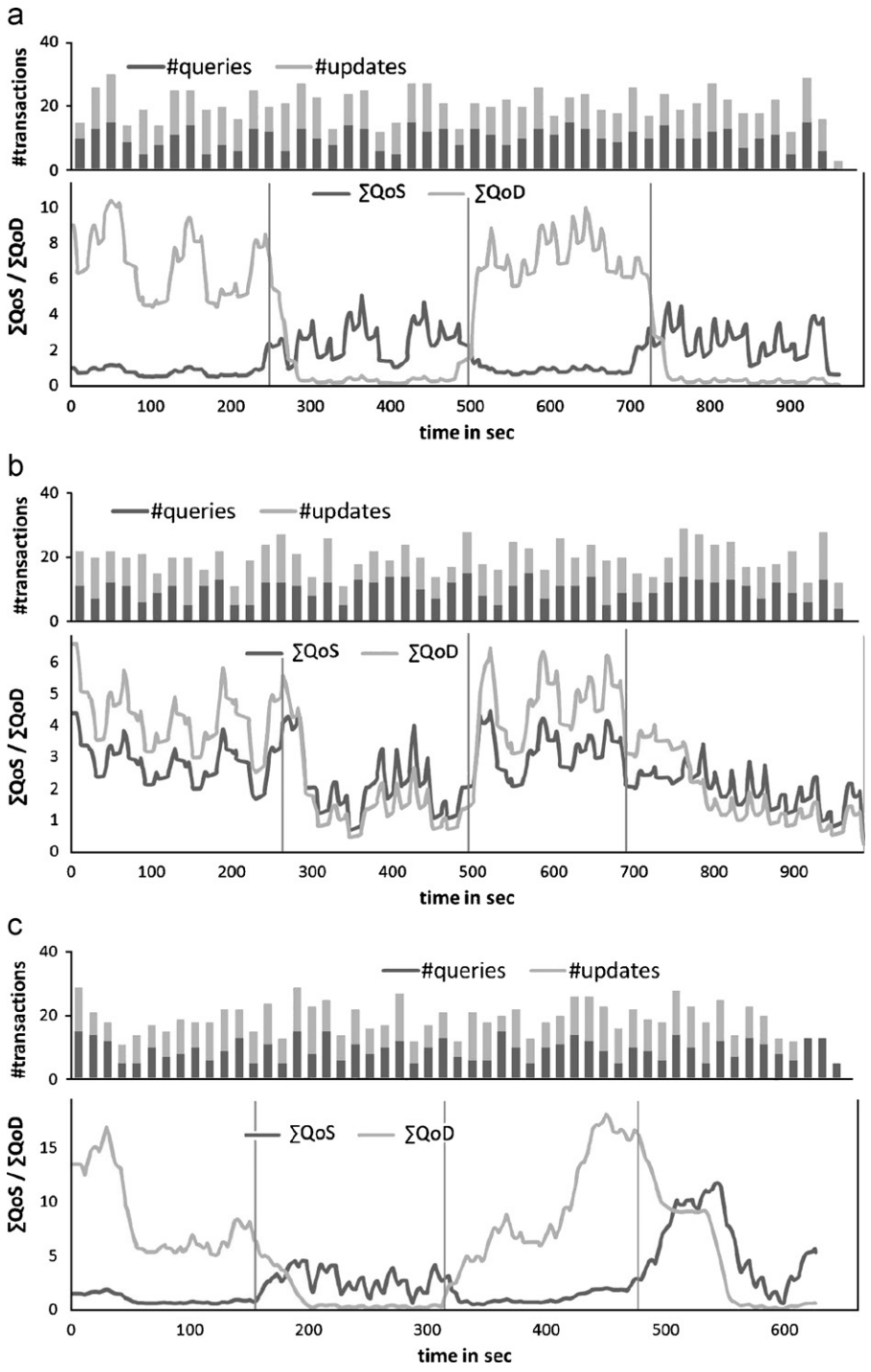

Fig. 11. Adaptability to user requirements for different workload settings on CDBS dataset: (a) *LOW* workload, $\langle QoS_1, QoD_1 \rangle \quad \langle 0.1, 0.9 \rangle$, (b) *LOW* workload, $\langle QoS_1, QoD_1 \rangle \quad \langle 0.4, 0.6 \rangle$, (c) *HIGH* workload, $\langle QoS_1, QoD_1 \rangle \quad \langle 0.1, 0.9 \rangle$.

complete our analysis on adaptability, we stressed our system with a *MEDIUM* and a *HIGH* workload (Figs. 10c and 11c). We see that the adaptation of the QoS and QoD sums to changing trends needs longer than under the *LOW* workload in Figs. 10a and 11a. The reason is that the data warehouse is not able to process all queries (and updates) in time, which leads to a decreasing adaptability at trend inversions, since more and more old queries interfere with

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

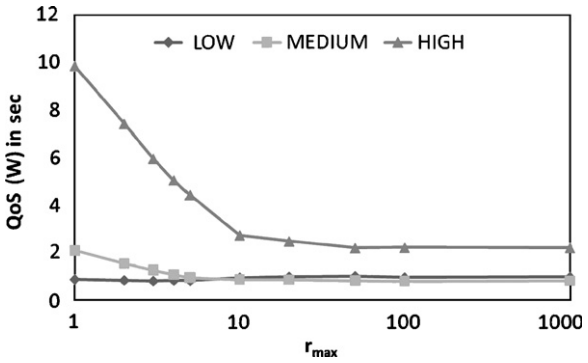

Fig. 12. QoS metric against maximum retention time (r_{max}) (TPC-DS dataset).

the overall QoS and QoD sums. Note that due to the strong turnarounds in trend in this experiment, this effect is particularly visible in both figures.

Note also that we applied a moving average with a window size of 30 queries to smoothen the data, which is very bursty in the original.

6.4. Significance of system parameter r_{max}

In Section 4.2.1, we saw that the system parameter r_{max} affects the retention time of a query in the system. With a rising r_{max}, the retention time of a query q increases because the delta by which the corresponding qos'_q values are incremented decreases. However, this is only the direct impact of r_{max} on one query. Fig. 12 plots the QoS metric for a random workload, which was executed 30 times, on the TPC DS dataset in dependence on an increasing r_{max}. In case of the *LOW* workload, we see that a growing r_{max} does not destabilize the QoS metric; for the *MEDIUM* workload, however, the metric decreases slightly, and for the *HIGH* workload, this trend becomes even more significant. The explanation is the following: the QoS metric considers only the queries with $qos_q \geqslant 0.5$ that means queries with a high demand for short retention times. With an increasing load and a constant r_{max}, these queries are increasingly displaced by queries with $qos_q < 0.5$, i.e., the QoS metric gets worse. For higher r_{max} values, queries advance at a slower pace in the query queue; hence, queries with $qos_q < 0.5$ remain at the tail of the query queue for a longer period, so their negative effect on the QoS metric is reduced.

Thus, the system parameter r_{max} allows us to specify the trade off between the overall QoS metric and the response time for queries with $qos_q < 0.5$. The retention time for this group of queries with high freshness demands ($qod_q > 0.5$) is not measured by our system metrics but should not be left unconsidered in practice.

7. Summary and future work

Living data warehouses have to manage continuous flows of updates and queries and must comply with conflicting requirements, such as short response times and data freshness. In this paper, we proposed a new and easy to use approach to combine both requirements in the presence of user preferences. We developed a two level scheduling algorithm called WINE that balances and prioritizes over updates and queries to maximize the user satisfaction. We compared WINE to three baseline algo rithms. Our evaluation showed that WINE performs better than the baseline algorithms under different workloads and changing trends in user requirements.

The scheduling algorithm represented here initially focused only on updating the base tables of a data warehouse and did not consider the ETL process or the maintenance of data marts. In the general case, the inclusion of the ETL process is not feasible, since the operators in the ETL process modify the schema (e.g., concatenation of attributes into a key attribute). That is to say, partitions as the smallest comparable units in our system model are not available in the ETL process. Thus, in order to be able to determine the partition a datum d of a data source belongs to, the complete ETL process would have to be run. In this case, it would be useless to even consider the scheduling of d, since d will have been propagated already. In the special case that the instances of the partitioning attributes (e.g., product group) are already known at the beginning of the ETL process, i.e., the datum d contains all required information, an extension of the system model to include the ETL process might be possible. The extension of the system model to cover the updating of materialized views, however, is possible in any case and represents a topic for our future work. The partitioning schema proposed here (see Section 5), which distributes data into partitions, i.e., into the smallest units to be considered, would additionally have to be applied to the data marts.

Furthermore, our proposed scheduling algorithm, WINE, would have to be extended from its current one level system model to a multi level system model. Initially, this could happen in a naive fashion by directly transferring the model, the two queues and the schedul ing algorithm to each data mart. For a more sophisticated solution, however, an additional cost model would have to be developed in order to be able to decide for queries which data mart would be most suitable for an efficient execution and whether or not it might even be preferable to execute them directly on the data warehouse. For example, it might be possible that for an environment, where multiple data marts store the same data or overlapping sets of data, queries with weak up to dateness ($QoD < 0.5$) requirements are directed to less frequently updated data marts, thus taking off some of the burden of the data marts that have to be updated rather frequently.

Aside from this extension of the system model, the processing of queries and updates will have to be transformed from the current sequential model to a parallel model (inter transaction parallelism). On the one hand, it may be possible for this purpose to map the user requirements to the CPU or memory resources to be used, and on the other hand, the matching information (derived from the partitioning) for queries and updates or for updates themselves may be used as additional input for the optimizer in order to use them for a more efficient

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Information Systems". 34 (4-5), S. 382-399. ISSN: 0306-4379.
https://doi.org/10.1016/j.is.2008.06.001

processing of transactions in parallel. Aside from that, additional implications, such as the maintenance of indices and potential locks for these DB objects, would have to be considered in the extended model.

In addition to the efforts spent with the model itself, the parameterization for different setups shall be analyzed, e.g., for warehouses with classic disk IO behavior, databases with solid state disks (no difference between sequential and random access any longer), OLAP main memory systems, etc. For each of these setups, guidelines shall be developed that describe how the scheduling for these individual special cases will have to be parameterized.

To summarize, we believe that the real time or push semantics introduced for data warehouses implicate an extended user model, which describes the varying user demands. In this paper, we proposed a very fundamental approach and developed auxiliary methods and algo rithms to support the prioritization of transactions based on user requirements. Extensions to a multi level system model, the shift towards parallel processing, and the analysis of the impact of different physical designs represent topics of our future work.

References

[1] J. Leung, L. Kelly, J.H. Anderson, Handbook of Scheduling: Algorithms, Models, and Performance Analysis, CRC Press, Inc., Boca Raton, FL, USA, 2004.

[2] H. Qu, A. Labrinidis, Preference-aware query and update scheduling in web-databases, in: ICDE, 2007, pp. 356–365.

[3] T. Thalhammer, M. Schrefl, M. Mohania, Active data warehouses: complementing olap with analysis rules, Data Knowl. Eng. 39 (3) (2001) 241–269.

[4] D.F. Ferguson, Y. Yemini, C. Nikolaou, Microeconomic algorithms for load balancing in distributed computer systems, in: ICDCS, 1988, pp. 491–499.

[5] D.F. Ferguson, C. Nikolaou, J. Sairamesh, Y. Yemini, Economic models for allocating resources in computer systems, in: Market-Based Control: A Paradigm for Distributed Resource Allocation, 1996, pp. 156–183.

[6] J.F. Kurose, M. Schwartz, Y. Yemini, A microeconomic approach to decentralized optimization of channel access policies in multiaccess networks, in: ICDCS, 1985, pp. 70–77.

[7] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, C. Staelin, An economic paradigm for query processing and data migration in mariposa, in: PDIS '94: Proceedings of the Third International Conference on Parallel and Distributed Information Systems, Los Alamitos, CA, USA, IEEE Computer Society Press, Silver Spring, MD, 1994, pp. 58–68.

[8] C.A. Waldspurger, W.E. Weihl, Lottery scheduling: flexible proportional-share resource management, in: OSDI '94: Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation, Berkeley, CA, USA, USENIX Association, 1994, p. 1.

[9] M.P. Wellman, A market-oriented programming environment and its application to distributed multicommodity flow problems, J. Artif. Intell. Res. 1 (1993) 1–23.

[10] H. Qu, A. Labrinidis, D. Mossé, Unit: user-centric transaction management in web-database systems, in: ICDE, 2006, p. 33.

[11] K.-D. Kang, Managing deadline miss ratio and sensor data freshness in real-time databases, IEEE Trans. Knowl. Data Eng. 16 (10) (2004) 1200–1216 Senior Member—Sang H. Son and Fellow—John A. Stankovic.

[12] K.-D. Kang, S.H. Son, J.A. Stankovic, T.F. Abdelzaher, A qos-sensitive approach for timeliness and freshness guarantees in real-time databases, in: ECRTS, 2002, pp. 203–212.

[13] J.R. Haritsa, M.J. Carey, M. Livny, Value-based scheduling in real-time database systems, VLDB J. 2 (2) (1993) 117–152.

[14] D. Hong, T. Johnson, S. Chakravarthy, Real-time transaction scheduling: a cost conscious approach, in: P. Buneman, S. Jajodia (Eds.), Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, May 26–28, 1993, ACM Press, New York, 1993, pp. 197–206.

[15] D. Agrawal, A.E. Abbadi, A. Singh, T. Yurek, Efficient view maintenance at data warehouses, in: SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, ACM, New York, 1997, pp. 417–427.

[16] L. Ding, X. Zhang, E.A. Rundensteiner, The mre wrapper approach: enabling incremental view maintenance of data warehouses defined on multi-relation information sources, in: DOLAP '99: Proceedings of the 2nd ACM International Workshop on Data Warehousing and OLAP, New York, NY, USA, ACM, New York, 1999, pp. 30–35.

[17] X. Zhang, L. Ding, E.A. Rundensteiner, Pvm: parallel view maintenance under concurrent data updates of distributed sources, in: DaWaK '01: Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery, London, UK, Springer, Berlin, 2001, pp. 230–239.

[18] M. Teschke, A. Ulbrich, Concurrent warehouse maintenance without compromising session consistency, in: DEXA, 1998, pp. 776–785.

[19] J. Chen, S. Chen, E.A. Rundensteiner, A transactional model for data warehouse maintenance, in: ER '02: Proceedings of the 21st International Conference on Conceptual Modeling, London, UK, Springer, Berlin, 2002, pp. 247–262.

[20] S. Chen, J. Chen, X. Zhang, E.A. Rundensteiner, Detection and correction of conflicting source updates for view maintenance, in: ICDE '04: Proceedings of the 20th International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society, Silver Spring, MD, 2004, p. 436.

[21] H. Guo, P.-A. Larson, R. Ramakrishnan, J. Goldstein, Relaxed currency and consistency: how to say "good enough" in SQL, in: SIGMOD, New York, NY, USA, ACM Press, New York, 2004, pp. 815–826.

[22] P.-A. Larson, J. Goldstein, J. Zhou, Mtcache: transparent mid-tier database caching in sql server, in: ICDE '04: Proceedings of the 20th International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society, Silver Spring, MD, 2004, p. 177.

[23] J. Zhou, P.-A. Larson, H.G. Elmongui, Lazy maintenance of materialized views, in: VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB Endowment, 2007, pp. 231–242.

[24] Y. Zhuge, H. García-Molina, J. Hammer, J. Widom, View maintenance in a warehousing environment, SIGMOD Rec. 24 (2) (1995) 316–327.

[25] K. Salem, K. Beyer, B. Lindsay, R. Cochrane, How to roll a join: asynchronous incremental view maintenance, SIGMOD Rec. 29 (2) (2000) 129–140.

[26] S. Agrawal, V. Narasayya, B. Yang, Integrating vertical and horizontal partitioning into automated physical database design, in: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, ACM, New York, 2004, pp. 359–370.

[27] M. Thiele, J. Albrecht, W. Lehner, Optimistic coarse-grained cache semantics for data marts, in: SSDBM '06: Proceedings of the 18th International Conference on Scientific and Statistical Database Management, Washington, DC, USA, IEEE Computer Society, Silver Spring, MD, 2006, pp. 311–320.

[28] D. Quass, J. Widom, On-line warehouse view maintenance, in: SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, ACM, New York, 1997, pp. 393–404.

[29] M. Teschke, A. Ulbrich, Concurrent warehouse maintenance without compromising session consistency, in: DEXA '98: Proceedings of the 9th International Conference on Database and Expert Systems Applications, London, UK, Springer, Berlin, 1998, pp. 776–785.

[30] R. Othayoth, M. Poess, The making of TPC-DS, in: VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB Endowment, 2006, pp. 1049–1058.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden