Scheduling with Freshness and Performance Guarantees for Web Applications in the Cloud

Yingying Zhu **Mohamed A. Sharaf** **Xiaofang Zhou**

School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, Australia
Email: (y.zhu3, m.sharaf, zxf)@uq.edu.au

Abstract

Highly distributed data management platforms (e.g., PNUTS, Dynamo, Cassandra, and BigTable) are rapidly becoming the favorite choice for hosting modern web applications in the cloud. Among other features, these platforms rely on data partitioning, replication and relaxed consistency to achieve high levels of performance and scalability. However, these design choices often exhibit a trade-off between performance and *data freshness*. In this paper, in addition to performance SLAs, we also perceive an application tolerance to data staleness as another requirement determining the end-user satisfaction and our goal is to strike a fine balance between both the quality of service (QoS) and quality of data (QoD) perceived by the end-user. Towards that, we propose scheduling policies and mechanisms for efficiently allocating the recourses at each replica node so that to meet the conflicting requirements of user queries and replica updates. Our experimental results show that employing our scheduling strategies for resource allocation can provide significant improvements in the overall system utility when compared to the existing ones.

Keywords: Web Database, Distributed Database, Cloud Computing, Scheduling, Consistency, SLA, Quality of Data, Quality of Service.

1 Introduction

In modern web applications, user satisfaction or positive experience determines the applications' success (and keeps the competitors "more than a click away" [17]). A fundamental requirement in such web applications is to consistently meet the user's expectations for page load time as expressed by a Service Level Agreement (SLA). An example of a simple SLA is a web application guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second [8].

Clearly, application SLAs place stringent response time requirements on data management platforms demanding a near realtime performance. Towards this, several data management techniques have been continuously improved in order to maximize the SLA satisfaction of web database transactions. Examples of such techniques include data caching[19], data prefetching [6], adaptive transaction scheduling [9, 17], etc. However, the continuous growth in database-driven web applications as well as the complexity of user requirements required re-thinking the traditional database solutions and resulted in a new generation of highly distributed database platforms especially designed to meet the ever stringent performance requirements expected by today's end user. Examples of such platforms include PNUTS [7], Dynamo [8], Cassandra [15], and BigTable [5].

Such platforms are expected to meet strict operational requirements in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable [8]. However, the design choices for these platforms often exhibit a trade-off between perceived performance and *data freshness*, which is the focus of this paper. In particular, most of such modern platforms share the following key design choices:

- *Data Partitioning and Replication:* Dynamically partitioning data across the available storage nodes allows the system to incrementally scale out, where adding capacity becomes as simple as adding new servers [7]. Further, in a large-scale web application, users are scattered across the globe which makes it critical to have data replicas on multiple continents for low-latency access [7]. For instance, Dynamo uses a synthesis of well known techniques for data partitioning using consistent hashing as well as data replication.
- *Key-Value Data Model:* Presents a simplified data model to the user based on a key-value data store motivated by the observation that the big majority of web applications only need primary-key data access manipulating one record at a time. For example, Dynamo provides simple `get()` and `put()` operations for the read and write to a data item that is uniquely identified by a key, while no operation can span multiple data items.

Clearly, the above features directly contribute to improving the performance of web applications and meeting the pre-specified SLA requirements. However, achieving *serializability* for web transactions over a globally-replicated and distributed system is very expensive and often unnecessary [7]. In particular, web applications expect and tolerate weaker levels of consistency. For instance, Dynamo is designed to be an *eventually consistent* data store; that is all updates reach all replicas eventually. Similarly, PNUTS provides a consistency model that is between general serializability and eventual consistency.

While providing weaker levels of consistency allow for high availability, this often comes at the expense of data freshness where user queries might access *stale* data. This is typically accepted by most

web applications only if the perceived staleness is bounded within some pre-specified staleness tolerance [10]. Meanwhile, current platforms cannot guarantee such bounds on data staleness while still satisfying the stringent performance SLAs. For instance, in the Dynamo platform, replica synchronization tasks are executed in the background at low priority so that to allow enough resources for running the foreground queries and meeting their SLAs.

In this paper, in addition to a query SLA, we also perceive an application tolerance to data staleness as another requirement determining the end-user satisfaction, and in turn the success of a web application. Towards this, our goal is to balance the trade-off between the perceived Quality of Service (QoS) as expressed by performance SLA and the Quality of Data (QoD) as expressed by freshness. Our approach towards achieving that goal relies on efficiently allocating the resources at each node so that to meet the conflicting requirements of the *foreground* user transactions and the *background* replica updates. In particular, in this paper we propose new schemes for scheduling the execution of both user transactions and replica updates for database-driven web applications in the cloud.

To this end, several research efforts have leveraged scheduling as a method for balancing the trade-off between QoS and QoD in contexts such as web databases [16], realtime databases [1, 13], and realtime data warehouses [2, 20, 12]. Those approaches adopt various high-level mechanisms as well as different low-level scheduling policies.

For instance, the *On-Demand (OD)* [1] mechanism *couples* the execution of pending replica updates with the arriving user requests, where all the data items read by a certain query are refreshed on demand before the execution of that query. This strategy has been shown to be beneficial in saving system resources [11]. However, the OD mechanism employs simple low-level policies for scheduling queries (and in turn updates) which fall short in meeting the requirements of modern web applications.

On the other hand, the *QUTS* [16] approach considers both the QoS and QoD requirements of web applications under a unified *Quality Contracts* [14] model. However, QUTS *decouples* the execution of queries from that of replica updates, where it allocates to each a separate time quota of the system resources. This allows QUTS to target general database transactions where it is not easy to determine the data objects read by a query beforehand. But at the same time, this decoupling might waste significant system resources.

To the contrary, our scheduling approach presented in this paper leverages the specific characteristics of modern key-value data stores towards satisfying the QoS and QoD specifications of web applications. In particular, we propose high-level mechanisms together with low-level scheduling policies that consider the "cost" and "benefit" of executing background replica updates in conjunction with the foreground user requests so that to improve both performance and freshness.

Moreover, between the two extremes of coupling or decoupling the scheduling of user reads and replica updates, we propose a new hybrid mechanism called *FIT*, which integrates the advantages of both in oder to maximize the system gains. Our experimental results show that employing our scheduling schemes for resource allocation can provide significant improvements in the overall system utility when compared to existing policies.

The rest part of this paper is organized as follows. Section 2 describes the system model. Our proposed scheduling policies are presented in section 3. Section 4 describes the evaluation environment. Section 5 presents our experimental results. Section 6 finalizes this paper with conclusions and future work.

2 System Model

We consider a distributed data management platform where data is partitioned and replicated across multiple nodes. The data replication model we adopt in this paper is very close to the one currently employed by the Yahoo! PNUTS [7]. Specifically, we assume that each data record has one master copy and multiple replicas. All the write operations on a certain record are directed to its master copy, then later propagated to the replicas. This propagation takes place in a *lazy* or *asynchronous* fashion, where a write is installed at the master first and updates are propagated in the background. This model allows for *timeline consistency* where all replicas will go through the same sequence of updates such that they will eventually converge to the latest update made by the application. Hence, a record read by an application might be stale unless it is the master copy or it is a replica that has already applied the latest update.

Under the model described above, a node in our system receives 3 types of record operations: 1) write to a master, 2) read to a master or replica, or 3) update to a replica. Our proposed policies operate at the node-level where they are responsible for scheduling the execution of those operations so that to maximize both QoS and QoD. Next, we describe the details of our model together with our metrics for QoS and QoD.

2.1 Database Replica

Each node typically stores a set of master records as well as another set of replica records. As described above, a master record is accessed by foreground read and write requests and it is always fresh. Whereas, a replica record is accessed by either a foreground read request (i.e., query) or a background refresh request (i.e., update) and it might be stale. In the rest of this discussion, we will focus on the latter set of records (i.e., replicas) since accessing these records leads to a trade-off between performance and freshness. In particular, we assume that a database replica B consists of M data records (or objects) $\{O_1, O_2, ..., O_M\}$ that are accessed simultaneously by both user queries and system updates.

As in [10], we use the term replica broadly to include saved data derived from some underlying source tables. As such, it could be a replica in the ordinary sense as in the distributed data management platforms described above. But also a replica could be a materialized view internal to the web application for efficient query processing. For instance, to support social networking applications, PNUTS stores materialized views as regular tables that are asynchronously maintained by the system [18].

Finally, a replica could be a web database which represents a portal updated aperiodically by external sources. For example, in a stock information application, external databases such as the New York Stock Exchange store the history of updates, whereas the web database corresponds to a snapshot view reflecting the most recent stock information as propagated by that external database [16, 1].

2.2 Updates

Updates to a data replica are queued internally in the node's *update queue* until they are scheduled for execution. Each update U_i has a timestamp t_i representing the time when it was generated at the master copy. Further, each update modifies the value of a single record at a time using an operation such as `set()` or `put()` to set the value of a replica record to the same value committed at the master copy at time t_i.

Finally, each update U_i is characterized by a cost C_{iu}, which reflects the time required for processing the update and installing it onto the replica. This processing time incorporates both CPU and I/O costs and is typically determined by monitoring the processing of previous updates over a reasonable time window.

2.3 Queries

In our model, each query Q_i represents a `get()` operation to the key-value data store. Rendering a web page in modern web applications typically fires a large number of such operations. For example, a page request to an e-commerce sites typically requires the rendering engine to construct its response by sending over 150 `get()` operations [8].

Similar to updates, in a key-value data store, each query Q_i will have a timestamp representing its arrival time A_i and will access a single record or data object O_i. Meanwhile, operations that touch multiple records simply require a component that generates multiple requests to individually access each record. Finally, C_{iq} denotes the cost for retrieving and processing that data record. Like update processing, query processing time incorporates both CPU and I/O costs and is statistically estimated over time.

2.4 QoS and QoD Metrics

There are several metrics for capturing the user perceived QoS as well as QoD. In this paper, we focus on QoS in terms of minimizing *tardiness* and QoD in terms of minimizing *staleness* and our goal is to strike a fine balance between both metrics across all the user queries submitted to the system.

Ideally, if a query Q_i finishes execution at time F_i, then F_i should be within the QoS tolerance of Q_i. Similarly, the staleness of a record O_i accessed by the query should be within its QoD tolerance. However, in the presence of multiple queries and updates competing for the system resources, Q_i might experience queuing delays or access stale data that fall beyond its tolerance.

The natural way to capture those deviations is to define for each query Q_i two deadlines: 1) Tardiness Deadline (D_i), and 2) Staleness Deadline (S_i). In our model, those two deadlines represent the QoS and QoD requirements of a query and violating either incurs a *penalty* to be paid by the system. In particular, to specify the QoS and QoD requirements, each query is associated with the following parameters:

- Weight (W_i): the weight assigned to query Q_i, which represents its importance to the system.
- QoS Factor (α_{is}): the fraction of the weight assigned to QoS, which represents the QoS importance to the application.
- QoD Factor (α_{id}): the fraction of the weight assigned to QoD ($= 1.0 - \alpha_{is}$), which represents the QoD importance to the application.

Table 1: Model Parameters

Parameter	*Symbol*
Database	B
Objects in Database	$\{O_1, ..., O_M\}$
Write-only Transaction: Update	U_i
Timestamp of First Unapplied Updates	R_i
Read-only Transaction: Query	Q_i
Arrive Time of Query	A_i
Finish Time of Query	F_i
Weight of each query	w_i
QoS Factor	α_{is}
QoD Factor	α_{id}
QoS Tolerance	γ_{is}
QoD Tolerance	γ_{id}
Tardiness Deadline	$D_i = A_i + \gamma_{is}$
Staleness deadline	$S_i = R_i + \gamma_{id}$

- QoS Tolerance (γ_{is}): the tolerance of query Q_i to tardiness in time units.
- QoD Tolerance (γ_{id}): the tolerance of query Q_i to staleness in time units.

2.4.1 Query Perceived Tardiness

The tardiness Deadline (D_i) is defined as:

$$D_i = A_i + \gamma_{is} \tag{1}$$

where A_i is the arrival time of query Q_i and γ_{is} is its tolerance as described above.

if Q_i cannot meet its deadline, the system will still execute it but it will be "penalized" for the delay beyond the deadline D_i. This penalty per query is known as *tardiness* which is formally defined as:

Definition 1 *Tardiness, T_i, for query Q_i is the total amount of time spent by Q_i in the system beyond its deadline D_i. That is, $T_i = 0$ if $F_i \leq D_i$, and $T_i = F_i - D_i$ otherwise.*

2.4.2 Query Perceived Staleness

The staleness deadline (S_i) is defined as:

$$S_i = R_i + \gamma_{id} \tag{2}$$

where R_i is the timestamp of the *first unapplied update* to data object O_i and γ_{id} is the tolerance of Q_i to staleness as described above.

In particular, at time R_i, a replica record O_i is rendered stale because of the generation of a new update at the master copy. If that update is still unapplied until the time Q_i is scheduled for execution, then the system will still execute the query but it will be penalized for the staleness beyond the deadline S_i. This penalty per query is known as *staleness* (or *age* [3]) which is formally defined as:

Definition 2 *Staleness, L_i, for query Q_i is the total amount of staleness accumulated by O_i beyond Q_i's staleness deadline S_i. That is, $L_i = 0$ if $F_i \leq S_i$, and $L_i = F_i - S_i$ otherwise.*

2.5 Problem Definition

Given the above definitions of tardiness and staleness, our goal is to minimize the total combined penalty incurred by the system. This combined penalty per query Q_i is simply the sum of weighted staleness and weighted tardiness (as shown in Figures 1, 2, and 3) and is computed as follows:

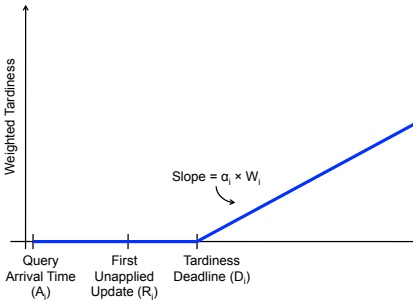

Figure 1: Weighted Tardiness

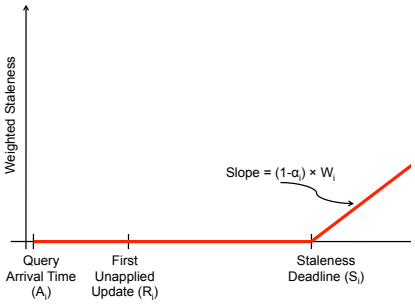

Figure 2: Weighted Staleness

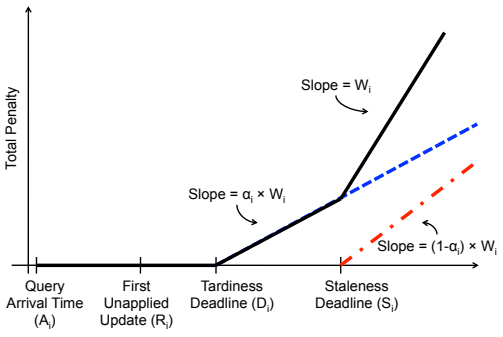

Figure 3: Total Penalty

$$P_i = W_i[(\alpha_{is} \times T_i) + (\alpha_{id} \times L_i)]$$

Or equivalently,

$$P_i = W_i[(\alpha_i \times T_i) + ((1 - \alpha_i) \times L_i)] \quad (3)$$

where $\alpha_i = \alpha_{is}$ and W_i is the weight of query Q_i as defined above.

Hence, the system objective is to minimize the *average penalty* which is defined as:

Definition 3 *The average penalty for N queries is:* $\frac{1}{N}\sum_{i=1}^{N} P_i$.

In the next section, we will discuss several query and update scheduling strategies for achieving the objective defined above.

3 Scheduling Strategies

In this section, we present several strategies for the scheduling of both pending queries and updates at each node in the system. For each strategy, we make the distinction between the high-level general mechanism specifying the dependency between queries and updates (e.g., coupled, decoupled, etc.) and the low-level scheduling policy used for ordering the execution of those queries and updates (e.g., FCFS, EDF, etc.). Specifically, we first propose low-level scheduling policies that extend the On-Demand mechanism [1] by considering query characteristics (Sections 3.1) as well as update characteristics (Section 3.2). Then, we propose our new *FIT* mechanism together with low-level scheduling policies that further enable balancing the trade-off between QoS and QoD (Section 3.3).

3.1 Query-aware On-Demand Scheduling

Recall that the *On-Demand (OD)* [1] mechanism *couples* the execution of the pending replica updates with the arriving user requests, where all the data items read by a certain query are refreshed on demand before the execution of that query. This mechanism is well suited for key-value data stores where each record is accessed by its key leading to a simple coupling between queries and updates.

Further, the OD mechanism also allows for minimizing the system resources needed to install replica updates in a timeline consistency system like PNUTS. To explain this, recall that in a key-value data store, updates are *blind* operations that do not require reading the current value of a record before updating it. Hence, in a single-master system like PNUTS, all replicas will go through the same sequence of blind updates such that they will eventually converge to the latest update made by the application. Accordingly, the arrival of a new update to a certain record will make any pending update to that same record worthless as in the *Thomas Write Rule* [21]. That is, a replica can converge simply by applying the newest update skipping any intermediate ones.

Given the advantages of the OD mechanism, we have decided to further investigate its underlying scheduling policies. In general, under OD, queries are always given precedence over updates. However, when a query Q_i encounters a stale data object O_i, the update queue will first be checked if there is a pending update to O_i (i.e., U_i). If an update is found, it is applied before executing the query. This provides an attractive property which is maximizing the freshness of data by applying any pending relevant updates first, which results in almost no penalty for data staleness in our system. However, in terms of QoS, the On-Demand mechanism suffers from a major drawback as it employs a basic *First-Come-First-Served (FCFS)* policy where the arrival time of query Q_i determines its priority. FCFS has been shown to perform very poorly under deadline-based metrics such as tardiness [4], which leads to high QoS penalties for our system that are expected to overweight the gains from improving QoD provided by the OD mechanism. Hence, we propose extending OD with a set of priority-based scheduling policies that are well known for performing reasonably well under deadlines.

For all of those policies, for each pending query Q_i we compute a priority V_i based on some of the properties of Q_i. For the query with the highest priority, we first apply the pending update (if any) to the data item O_i, then execute the query as in the On-Demand mechanism. We first start with the FCFS-Q policy.

FCFS-Q: First-Come-First-Served (FCFS) has been proposed as the scheduling policy under the OD mechanism [1]. Under FCFS, each query Q_i is assigned a priority $V_i = \frac{1}{A_i}$, where A_i is the arrival time of query Q_i as described in Section 2. FCFS is a fair scheduling policy since it bounds the waiting time of a query in the system queue. However, this is often at odds with minimizing system performance metrics such as response time or tardiness.

EDF-Q: Earliest Deadline First (EDF) is one clear alternative for replacing FCFS under the OD mech-

anism. Under EDF, each query Q_i is assigned a priority $V_i = \frac{1}{D_i}$, where D_i is the tardiness deadline of query Q_i as described in Section 2. It has been shown that EDF provides a close to zero tardiness under low to medium system utilization which makes it attractive for web database during periods of light workload.

WSJF-Q: Weighted Shortest Job First (WSJF) is another alternative under the OD mechanism as it considers both the query processing time and its weight. Here, we only need to consider the fraction of weight pertaining to QoS (i.e., $\alpha_i W_i$) since the QoD component of weight is already maximized under the OD mechanism. Hence, under WSJF-Q, each query Q_i is assigned a priority $V_i = \frac{\alpha_i W_i}{C_{iq}}$, where $\alpha_i W_i$ is the QoS weight component and C_{iq} is processing of query Q_i as described in Section 2. It has been shown that WSJF minimizes the weighted tardiness under high system utilization as opposed to EDF which might exhibit a "domino effect" [9]. This makes WSJF especially attractive for web database during periods of high workload which is expected to be the norm for the applications we are considering in this work.

Density-Q: The density policy is very similar to the WSJF-Q except that it considers the query benefit (or penalty) at the current time rather than its weight [12]. As in WSJF-Q, we only need to consider the fraction of penalty to pertaining QoS since the QoD component of penalty is already minimized under the OD mechanism. Hence, under Density-Q, each query Q_i is assigned a priority $\frac{-W_i \alpha_i \times (\tau + C_{iq} - D_i)^+}{C_{iq}}$ where τ is the current time where a scheduling decision is to be made and $\tau + C_{iq}$ is the time where the query finishes execution and $(\tau + C_{iq} - D_i)^+$ is the positive value of the term, i.e., $max(0, \tau + C_{iq} - D_i)$.

If $\tau + C_{iq} \leq D_i$, then Q_i will finish before its deadline and the system incurred penalty is 0. However, if $\tau + C_{iq} > D_i$, then Q_i will miss its deadline and the system incurred penalty is the weighted tardiness $W_i \alpha_i \times (\tau + C_{iq} - D_i)$ or equivalently, the system benefit is the negative of that value as reflected in the priority function. Note that in comparison to WSJF-Q, as a result of considering current penalty rather than just weight, Density-Q recognizes the tardiness deadline as a *critical point* where a query starts accumulating tardiness leading to system penalty.

It is worth mentioning that under all the policies above, the execution order of updates is determined by the execution order of queries. Hence, there is no need for an update scheduling policy and the scheduling decision is solely based on the query characteristics. Thus, we will call such strategy *Query-aware* and we denote the policies using "-Q" such as FCFS-Q. Also note that during intervals of light load where queries are more sporadic, updates could be scheduled independently. Specifically, if the query queue is empty, the system starts executing updates until a new query arrives. To schedule those updates we use the basic Shortest Job First (SJF) policy where each update is assigned a priority $V_i = \frac{1}{C_{iu}}$. In the next section, we propose policies that further integrates the characteristics of the pending updates under the OD mechanism.

3.2 Update-aware On-Demand Scheduling

In the previous section, we have applied two features of the On-Demand approach, namely:

1. Applied any pending update to a data object before it is accessed by a query, and
2. Employed scheduling policies that only consider the properties of pending queries.

The first feature above enforces the On-Demand mechanism where updates are applied when an object is accessed leading to fresh data. Meanwhile, the second feature simplifies the scheduling decision by restricting the priority functions to only the query parameters. However, exploiting only the query parameters in scheduling might have a serious negative impact on the system performance.

In particular, all the policies presented above are oblivious to the properties of updates which might be in conflict with the properties of the corresponding query. For instance, under the WSJF-Q, if a query Q_i has the lowest processing cost then it might be selected for execution first regardless of the cost for refreshing data object O_i (i.e., C_{iu}). If that cost of installing the update happened to be very high, then all pending queries will be delayed and accumulating tardiness resulting in a poor overall system performance.

To avoid such conflict, we propose an *Update-aware* strategy, which works like the original On-Demand but employs scheduling policies that consider the characteristics of updates in addition to those of queries. Before explaining those policies, note that the negative impact of an update on the system is restricted to the QoS perceived by other queries but not on the perceived QoD. In particular, processing a certain update U_i with cost C_{iu} leads to delaying the processing of other queries and might lead to an increase in tardiness if those queries are close to their deadlines. However, it has no impact on the QoD under the On-Demand mechanism since data objects are always refreshed before accessed by a query leading to maximum freshness.

Hence, under the Update-aware version, we only need to modify those scheduling policies that consider processing cost, to include the cost of processing an update in addition to that of processing a query. Specifically, the EDF and FCFS policies will remain the same under update-aware, whereas we need new versions of WSJF and Density. For those two policies, for each pending query Q_i we compute a priority V_i based on some of the properties of Q_i and its corresponding U_i (if any). For the query-update pair with the highest priority, we first apply the pending update (if any) to the data item O_i, then execute the query as in the On-Demand mechanism.

WSJF-QU: Under the update-aware WSJF-QU, each query Q_i is assigned a priority

$$V_i = \frac{\alpha_i W_i}{C_{iq} + C_{iu}} \tag{4}$$

where $\alpha_i W_i$ is the QoS weight component, C_{iq} is the cost of processing query Q_i and C_{iu} is the cost of refreshing data object O_i by applying the pending update U_i.

Density-QU: Under the update-aware Density-QU, each query Q_i is assigned a priority

$$\frac{-W_i \alpha_i \times (\tau + C_{iq} + C_{iu} - D_i)^+}{C_{iq} + C_{iu}} \tag{5}$$

where τ is the current time where a scheduling decision is to be made.

Intuitively, the two policies above consider the negative impact of applying an update in terms of delaying other queries by an amount of time equal to the update cost C_{iu}. Further, the Density-QU policy also considers the negative impact of an update U_i on its own query Q_i since waiting until an update is installed might lead to Q_i missing it tardiness deadline, which results in QoS penalty to the system.

3.3 FIT Scheduling

The On-Demand mechanism for scheduling asynchronous updates defers applying an update as much as possible (i.e., until a query request is about to access a stale data object). For "blind" updates, this allows for saving system resources that otherwise would have been unnecessarily wasted on installing intermediate updates. However, it is often the case that applying the most recent update is not that necessary. This occurs under different conditions such as when the staleness of a data object is within the query's tolerance or in the extreme case when a query actually does not assign any weight to the QoD.

Even when the staleness violates the query requirement, applying an update might require high processing cost that will have a negative impact on the tardiness of that query and all the other pending queries on that node leading to an overall lower QoS. Towards this, we propose a new Freshness/Tardiness aware mechanism called *FIT* for the scheduling of queries and updates.

FIT, like OD, defers refreshing an object until it is requested by a query. However, under FIT, the scheduling policy reasons about the global impact of applying the update in terms of the utility of processing that update to the query under consideration as well as the other queries in the system. Before describing the details of our scheduling policies under FIT, we first introduce the general mechanism shared by all those policies. Specifically, under FIT, for each pending query Q_i, we compute two priorities:

1. v_i^+: The priority of Q_i if it is executed together with the latest corresponding update U_i (if any), and
2. v_i^-: The priority of Q_i if it is executed while *"skipping" U_i*.

Finally, $Q_i's$ priority V_i is computed as:

$$V_i = max(v_i^-, v_i^+)$$

For the query with the highest priority V_i, if $v_i^+ > v_i^-$, then first apply the pending update (if any) to the data object O_i, then execute the query. Otherwise, the query will directly access the stale data object O_i and the pending update U_i will not be removed from the updates queue.

In order to understand the intuition underlying each of the next scheduling policies, recall that the priority v_i^+ corresponds to only a QoS penalty as represented in Figure 1, whereas a priority v_i^- corresponds to a combined QoS and QoD penalty as represented in Figure 3. Hence, the v_i^+ priority under all scheduling policies is the same as their counterparts under the update-aware mechanisms, whereas the v_i^- priority should reflect the impact of skipping an update in those cases where it is more beneficial than applying it (i.e., $v_i^- > v_i^+$).

Finally, note that measuring the impact of skipping or applying an update pertains only to those scheduling policies with priority functions that can capture that impact, namely, WSJF and Density, but not the EDF or FCFS policy.

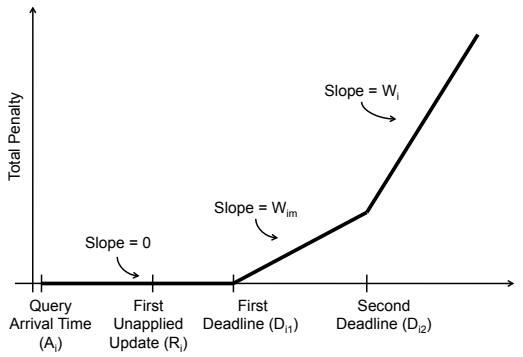

Figure 4: General Penalty Function under FIT

In order to compute v_i^-, we need to consider the impact of an update on both QoS and QoD. From Figure 3, we notice that combined penalty is a function in time with two *critical* points: 1) tardiness deadline, and 2) staleness deadline. In particular, assume the case in Figure 3 where $D_i < S_i$, then the penalty is zero up to time $\tau = D_i$ then it increases linearly with slope $\alpha_i \times W_i$ reflecting the penalty incurred by the system for not meeting the tardiness deadline. This slope stays the same up until time $\tau = S_i$ where the slope increases to be W_i reflecting the combined penalty for both staleness and tardiness. This slope remains constant until the query is eventually answered. In the opposite case where $D_i > S_i$, the penalty function will have the same general shape except that the first slope will be $(1 - \alpha_i) \times W_i$.

In general, we can represent the penalty as a function (Figure 4) with two deadlines: D_{i1}, and D_{i2} and three segments with slopes that have the following values: (a) 0, if $\tau \leq D_{i1}$, (b) W_{im}, if $D_{i1} < \tau \leq D_{i2}$, and (c) W_i, if $\tau > D_{i2}$. W_{im} is the *intermediate* weight when the query misses one of its deadlines but not both. As such, if $D_i < S_i$, then $W_{im} = \alpha_i \times W_i$ and if $D_i > S_i$, then $W_{im} = (1-\alpha_i) \times W_i$, whereas W_i is the weight when a query misses both its deadline as described in Section 2.

WSJF-FIT: Under WSJF-FIT, v_i^+ is computed similar to the update-aware counterpart, whereas v_i^- is computed based on Figure 4 as explained above.

$$v_i^+ = \frac{\alpha_i W_i}{C_{iq} + C_{iu}},\ v_i^- = \begin{cases} \frac{W_{im}}{C_{iq}} & \tau \leq D_{i1} \\ \frac{W_i}{C_{iq}} & \tau > D_{i1} \end{cases} \quad (6)$$

By considering the query cost C_{iq} in v_i^-, WSJF-FIT, Like WSJF-Q, also captures the negative impact of running a certain query Q_i on the other pending queries on the node. Similarly, it also captures the system loss in QoS if Q_i were to miss its tardiness deadline, which is expressed by the QoS portion of its weight. But in addition to that, it also captures the system loss in QoD if Q_i were to access a stale data object, which is expressed by the QoD portion of its weight.

To reflect the loss in QoS and QoD in v_i^-, we simply extended the basic WSJF policy to consider

two deadlines (Figure 4) instead of one deadline (Figure 1). In general, we can argue that WSJF sets the weights according to the slope of the next *critical point*. Hence, under WSJF-Q and WSJF-QU (Figure 1), at any time τ the weight will have only one value because there is only one critical point. However, under WSJF-FIT (Figure 4), after crossing the first critical point (i.e., D_{i1}) the weight is updated to reflect the future penalty incurred by the system if the query were to be delayed further, where that penalty is expressed by the slope at the next critical point (i.e., D_{i2}).

Density-FIT: Under Density-FIT, v_i^+ is computed similar to the update-aware counterpart, whereas v_i^- is computed to reflect the impact of having two deadlines as defined below:

$$v_i^+ = \frac{-W_i\alpha_i \times (\tau + C_{iq} + C_{iu} - D_i)^+}{C_{iq} + C_{iu}}, \qquad (7)$$

$$v_i^- = \frac{-W_{im}(\tau + C_{iq} - D_{i1})^+ - (W_i - W_{im})(\tau + C_{iq} - D_{i2})^+}{C_{iq}}$$

By balancing the trade-off between the "cost" and "benefit" of applying a replica update, FIT is able to strike a fine balance between QoS and QoD as we show in the next sections.

4 Experimental Evaluation

Testbed: We have created a simulator that implements the different mechanisms and policies discussed in this paper. The simulator takes as an input the system parameters, and generates the queries and updates based on these parameters such as deadlines, processing cost, etc. We have varied the parameters settings and conducted several experiments to test the performance of our proposed mechanisms and policies and compared them to other existing approaches.

Queries: For each simulated point, we generated 5000 queries where the data object accessed by each query is generated according to uniform distribution over the range[1, 100]. The processing cost C_{qi} for each query Q_i depends on the accessed data object and is generated according to a uniform distribution over the range [10, 50] mSec. Each query Q_i is assigned a tardiness tolerance $\gamma_{is} = k_i * C_{iq}$, where k_i is generated uniformly over the range $[1, k_{max}]$. Hence, the tardiness deadline $D_i = A_i + k_i * C_{iq}$, where we set $k_{max} = 5$ in our experiments.

Each query Q_i is also assigned a staleness deadline S_i, which is related to the tardiness deadline of the query (i.e., D_i). This enables us to control the distance between the 2 deadlines for staleness and tardiness. In our experiments, S_i is generated using uniform distribution in the range $[D_i + \lambda_l, D_i + \lambda_r]$. In the default setting, $\lambda_l = -50$ and $\lambda_r = +50$. Note that if $S_i < R_i$, then we set $S_i = R_i$. That is, the staleness deadline has to be at least equal to the arrival time of the last unapplied update but not less so that to reflect only positive values of tolerance.

To specify the QoS and QoD requirement, each query is assigned a weight W_i uniformly distributed over the range [1,10] which represents the importance of that query. The QoS fraction of the weight (i.e., α) is set in the range [0.1–α_{max}] where in the default setting $\alpha_{max} = 1.0$ and the skewness for α's zipf distribution is 0.0 (i.e., uniform). The arrival of queries is modeled as a poisson process, where we vary the arrival rate of queries between 10 to 50 queries/sec. Given our distribution for processing costs, an arrival rate of 50 queries/second is equivalent to $\simeq$ 100% utilization of the replica node.

Table 2: Simulation Parameters

Parameter	*value*
Number of Data Objects	100
Number of Queries	5000
Data Object Access Cost	Uniform over[10, 50]
Update Cost	Zipf over $[10, C_{max}]$
Query Arrival Rate	5–50
Update Arrival Rate	50
Query Deadline Parameter	$k_{max} = 5$
Importance Weight	Uniform over [1, 10]
QoS fraction α	Uniform over [0.1, 1.0]

Table 3: Mechanisms and Policies

	Query-aware	**Update-aware**	**FIT**
FCFS	FCFS-Q		
EDF	EDF-Q		
WSJF	WSJF-Q	WSJF-QU	WSJF-FIT
Density	Density-Q	Density-QU	Density-FIT

Updates: The processing cost C_{ui} of each update U_i is also generated according to a $Zipf$ distribution over the range $[10, C_{max}]$ mSec. Varying the values of C_{max} and the skewness $Zipf$ allows us to control the impact of updates on the system load. In the default setting, C_{max} is set to 100 with the default $Zipf$ parameter for skewness θ_u set to 0.5 and skewed towards the high-end of the cost range. The arrival of updates is modeled as a poisson process, where we set the arrival rate to 50 updates/sec.

Table 2 summarizes our simulation parameters and their default values.

Algorithms: Table 3 summarizes the mechanisms and policies discussed in this paper and simulated in our experiments. A blank entry in Table 3 entails that the corresponding policy is not applicable under the mechanism.

Additionally, we have also included the $QUTS$ policy [16]. QUTS prioritizes the scheduling of updates and queries using a two-level scheduling scheme that dynamically allocates CPU resources to updates and queries according to user preferences on QoS (tardiness) and QoD (staleness). To implement QUTS under our model, we have changed the hard deadlines to be soft deadlines and changed the QoS and QoD functions to our tardiness and staleness metric. We have also replaced the user's preference on QoS and QoD by α_{is} and α_{id}. Finally, we have set the the atom time τ to 0.01 and the adaptation period to 1 time unit.

5 Experimental Results

5.1 Impact of Query Arrival Rate

In this experiment, we set all the parameters to the default values mentioned in the previous section. We varied the query arrival rate from 5 queries/second to 50 queries/second. Under our setting for query processing costs, an arrival rate of 50 queries/second will bring the node up to a utilization around 100%.

Figure 5 shows the penalty incurred by the system when applying different strategies for scheduling queries and updates. The schedulers included in this

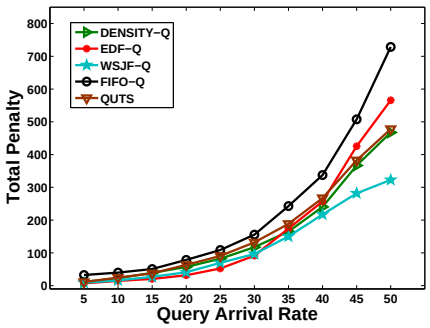

Figure 5: Query-aware Policies + QUTS

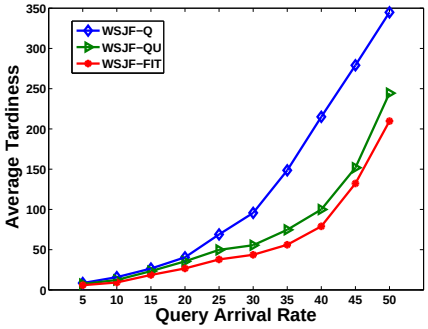

Figure 7: Average Tardiness of WSJF

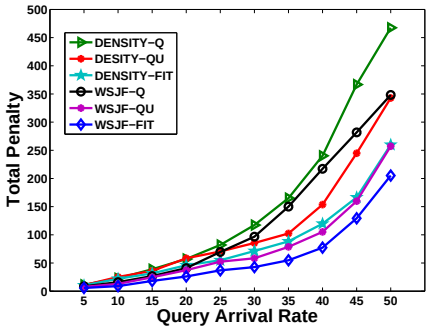

Figure 6: Comparison of All Mechanisms

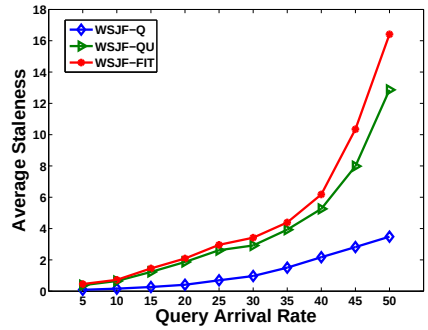

Figure 8: Average Staleness of WSJF

figure are the query-aware ones presented in Section 3 in addition to QUTS as described in Section 4. Figure 5 shows that in general, for all schedulers the penalty increases with increasing the query arrival rate (i.e., increasing utilization). However, FCFS (employed by OD) exhibits the highest penalty. Meanwhile, the performance of EDF came as expected where it provided the lowest penalty at low utilization but that penalty increased significantly at high utilization because of the mentioned domino effect.

Additionally, Figure 5 also shows that at high query arrival rate, the performance of QUTS is very similar to that of Density, whereas WSJF outperforms them both. Specifically, at the arrival rate of 50 queries/sec, WSJF-Q reduces the system penalty by 35% compared to QUTS, 33% compared to Density-Q, and 57% compared to FCFS-Q. Compared to QUTS, WSJF-Q couples the scheduling of queries and updates which allows for providing near maximum QoD, while saving the system resources by applying the latest updates. Finally, one might be surprised that WSJF-Q outperforms Density-Q when the latter considers both the query tardiness deadline and processing cost. However, the reason for that is that WSJF-Q considers the slope of the penalty function, whereas Density-Q considers the instantaneous value of the penalty function. Hence, Density-Q might favor a query that seems to currently incur high instantaneous penalty over another query that will incur higher penalty in the future which is expressed via a high slope and is recognized by the WSJF-Q policy.

In Figure 6, we focus on comparing the performance of the different mechanisms discussed in this paper. In particular, we use the same settings for the results shown in Figure 5, but we include only the Density-Q and WSJF-Q policies because they provided the best performance under the query-ware mechanism (as illustrated in Figure 5). Additionally, we have also included their update-aware counterparts (i.e., Density-QU and WSJF-QU) and the FIT counterparts (i.e., Density-FIT and WSJF-FIT).

Figure 6 shows that for both the Density and the WSJF, the update-aware version performs better than the only query-aware version and that the FIT version performs better than both the query-aware and the update-aware. For instance, at query arrival rate of 50 queries/sec, WSJF-FIT reduces the system penalty by 67% compared to WSFJ-Q and by 22% compared to WSFJ-QU. These gains are further depicted in Figures 7 and 8 where we break down the penalty incurred by the system into its two components: tardiness and staleness.

Figure 7 shows the tardiness penalties (i.e., loss in QoS) for the WSJF policies. WSJF-FIT exhibits the lowest loss in tardiness since it might selectively decide to skip some updates if the benefit of an update does not justify its cost, thus saving resources that might be needed by other queries and updates. Figure 7 shows that at 50 queries/sec, WSJF-FIT reduces the tardiness penalty by 37% vs. WSJF-Q and 18% vs. WSJF-QU.

As expected, the gains provided by WSJF-FIT in reducing the tardiness penalty come at the expense of an increase in the staleness penalty as shown in Figure 8. The figure shows that by skipping some updates, WSJF-FIT increased the staleness penalty compared to both WSJF-Q and WSJF-QU. However, these losses are countered by higher gains in terms of reducing the tardiness penalty leading to striking a fine balance between QoS and QoD as previously shown in Figure 6.

5.2 Impact of different QoS and QoD preferences α

To further illustrate the trade-off between QoS (i.e., tardiness) and QoD (i.e., staleness), in this experiment we keep the same default values as in the pre-

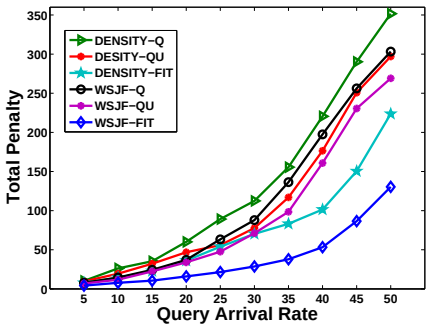

Figure 9: Comparison of All Mechanisms

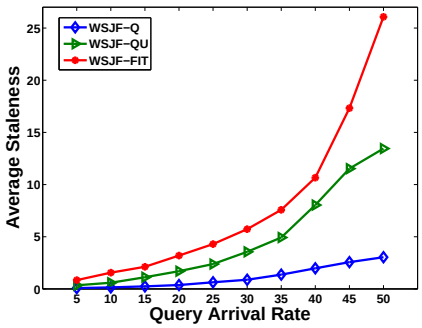

Figure 11: Average Staleness of WSJF

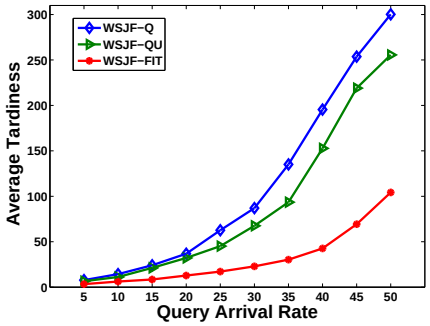

Figure 10: Average Tardiness of WSJF

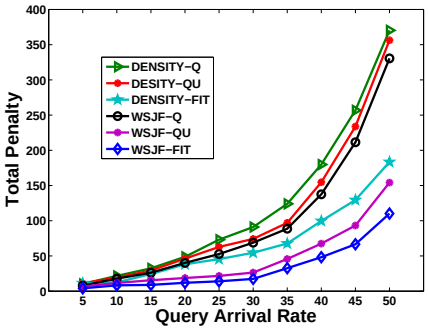

Figure 12: Comparison of All Mechanisms

vious one except that we increase the skewness of α's zipf distribution from 0.0 to 1.7 which is skewed towards high values of α. This leads to more queries giving higher preference to QoS over QoD, which in turn results in the system being penalized more for violating tardiness deadlines than staleness deadlines. Figure 9 shows our experimental results under that setting.

Figure 9 clearly highlights the benefits achieved by the Density and WSJF policies under the FIT mechanism vs. their counterpart under both the query-aware and update-aware mechanisms. For instance, in Figure 9 at query arrival rate of 50 queries/sec, WSJF-FIT reduces the system penalty by 59% compared to WSFJ-Q and by 53% compared to WSFJ-QU (vs. only 37% and 18% under the settings for Figure 6). This increase in gain (or equivalently reductions in penalty) is due to WSJF-FIT dynamically skipping updates that correspond to queries with low weight for QoD and are more interested in QoS as expressed by the α setting. This trade-off is further illustrated in Figures 10 and 11, where we break down the penalty incurred by the system into its two components of tardiness and staleness.

5.3 Impact of update cost (θ_u)

In this experiment, we kept the same default values as in the first experiment except that we increased the skewness of the update cost (i.e., $C_{u,i}$) zipf distribution from 0.5 to 1.7 which is skewed towards high values over the range $[10, C_{max}]$ where $C_{max} = 100$. This leads to more updates being expensive and requiring more system resource to refresh the stale data. Figure 12 shows our experimental results under that setting.

Figure 12 illustrates that under this setting, WSJF-FIT still outperforms all of the other policies. However, under this setting the reduction in penalty provided by WSJF-FIT vs. WSJF-QU is only 31% (in comparison to 53% in Figure 9). The reason for that closer gap in performance is that under this setting, WSJF-QU (being update-aware) will also recognize those updates with high processing costs and give them lower priority to favor queries and updates with lower costs. Similarly, WSJF-FIT will recognize those expensive updates and will either skip them (if they have low benefit) or give them low priority (if they have high enough benefit to balance the high cost). This trade-off is further illustrated in Figures 13 and 14, where we break down the penalty incurred by the system into its two components of tardiness and staleness.

6 Conclusions and Future Work

Motivated by the need for providing guarantees on both query performance and data currency in highly distributed data management platforms, we addressed the problem of scheduling queries and updates to strike a fine balance between QoS and QoD. Towards this, we presented three mechanisms for the scheduler implementation together with scheduling policies that work in conjunction with those mechanisms. Our experimental results show the the FIT mechanism introduced in this paper, together with the WSJF-FIT policy can efficiently allocate the available resources across queries and updates to maximize the system utility.

While our proposed scheduler is designed to operate at the node-level, in the future we plan to investigate global solutions that work at the system-level for achieving further improvements in performance. We are also planning to investigate advanced query/update scheduling policies that dynamically adapt to the workload and provide the best performance under both low and high utilizations.

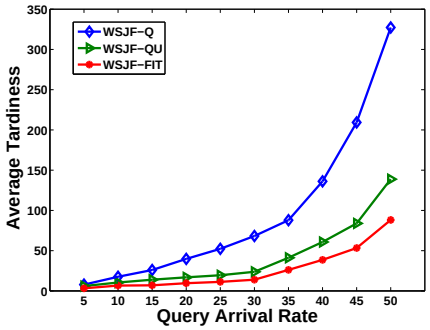

Figure 13: Average Tardiness of WSJF

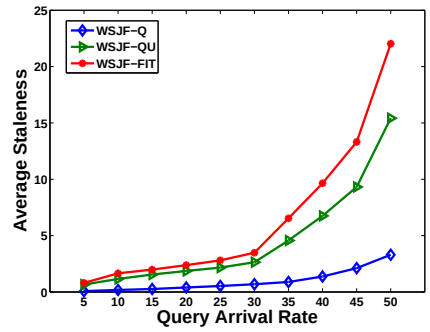

Figure 14: Average Staleness of WSJF

Acknowledgements: We would like to thank the anonymous reviewers for their thoughtful and constructive comments. We also highly appreciate the feedback we have received from Panos K. Chrysanthis and the students of the CS 3551 seminar course at the Univ. of Pittsburgh.

References

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *SIGMOD Conference*, pages 245–256, 1995.

[2] M. Bateni, L. Golab, M. T. Hajiaghayi, and H. J. Karloff. Scheduling to minimize staleness and stretch in real-time data warehouses. In *SPAA*, pages 29–38, 2009.

[3] M. Bouzeghoub and V. Peralta. A framework for analysis of data freshness. In *IQIS*, pages 59–67, 2004.

[4] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *RTSS '95*.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[6] X. Chen and X. Zhang. Coordinated data prefetching for web contents. *Computer Communications*, 28(17):1947–1958, 2005.

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[9] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Adaptive scheduling of web transactions. In *ICDE*, pages 357–368, 2009.

[10] H. Guo, P.-Å. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: How to say "good enough" in sql. In *SIGMOD Conference*, pages 815–826, 2004.

[11] T. Gustafsson and J. Hansson. Dynamic on-demand updating of data in real-time database systems. In *SAC*, pages 846–853, 2004.

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

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

[14] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterprises. In *BIRTE*, pages 143–156, 2006.

[15] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *PODC*, page 5, 2009.

[16] H. Qu and A. Labrinidis. Preference-aware query and update scheduling in web-databases. In *ICDE*, pages 356–365, 2007.

[17] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and C. Amza. Optimizing i/o-intensive transactions in highly interactive applications. In *SIGMOD Conference*, pages 785–798, 2009.

[18] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD Conference*, pages 831–842, 2010.

[19] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. Analysis of caching and replication strategies for web applications. *IEEE Internet Computing*, 11(1):60–66, 2007.

[20] M. Thiele, A. Bader, and W. Lehner. Multi-objective scheduling for real-time data warehouses. In *BTW*, pages 307–326, 2009.

[21] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[22] M. Xiong, S. Han, K. yiu Lam, and D. Chen. Deferrable scheduling for maintaining real-time data freshness: Algorithms, analysis, and results. *IEEE Trans. Computers*, 57(7):952–964, 2008.