FAS — a Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components

Uwe Röhm Klemens Böhm* Hans-Jörg Schek Heiko Schuldt

Swiss Federal Institute of Technology
ETH Zentrum, 8092 Zurich, Switzerland
{roehm,boehm,schek,schuldt}@inf.ethz.ch

Abstract

Data warehouses offer a compromise between freshness of data and query evaluation times. However, a fixed preference ratio between these two variables is too undifferentiated. With our approach, clients submit a query together with an explicit *freshness limit* as a new Quality-of-Service parameter. Our architecture is a cluster of databases. The contribution of this article is the design, implementation, and evaluation of a coordination middleware. It schedules and routes updates and queries to cluster nodes, aiming at a high throughput of OLAP queries. The core of the middleware is a new protocol called FAS (*Freshness-Aware Scheduling*) with the following qualitative characteristics: (1) The requested freshness limit of queries is always met, and (2) data accessed within a transaction is consistent, independent of its freshness. Our evaluation shows that FAS has the following nice properties: OLAP query-evaluation times are close (within 10%) to the ones of an idealistic setup with no updates. FAS allows to effectively trade 'up-to-dateness' for query performance. Even when all queries request fresh data, FAS clearly outperforms synchronous replication. Finally, mean response times are independent of the cluster size (up to 128 nodes).

* Current affiliation: Otto-von-Guericke-Universität Magdeburg, Germany

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

1 Introduction

Data warehouses are closely tied to OLAP, i.e., online analytical processing of the vast amount of data of an organization. They typically offer a compromise between freshness of data and warehouse maintenance costs. Different application scenarios and users however have different preferences in this respect, and a fixed preference ratio is too undifferentiated. With our approach, clients submit a query together with an explicit *freshness limit* as a new Quality-of-Service parameter. In other words, readers may decide infinitely variable how much up-to-date the data accessed should be. The goal is to use this additional information to improve throughput. The concern of this article is the throughput of a stream of OLAP queries, i.e., we assume a read-mostly environment with many concurrent readers. This complements recent work on replication in OLTP scenarios, e.g., [6, 17].

The object of this study is a cluster of databases [18, 14]: this is a cluster of commodity computers, each node running an off-the-shelf database management system as transactional storage layer. This paper assumes that all cluster nodes are homogeneous, i.e., they run the same DBMS with the same database schema. Each node holds a full copy of the database, but the freshness of these copies may vary between cluster nodes. Finally, we assume that there is a coordination middleware layer on top of the cluster (cf. Figure 1). Clients submit query or update transactions to this middleware, instead of directly communicating with specific cluster nodes. The middleware schedules and routes updates and queries to cluster nodes. The *scheduler* generates a correct interleaved execution order. In general, scheduling allows for several cluster nodes where a query may execute. The *router* chooses one of these nodes for each query. The challenge with such a cluster architecture is (1) to achieve high performance with regard to the OLAP query stream, (2) to guarantee global correctness at the same time, and (3) to satisfy the freshness

requirements of all queries. This goes beyond existing replication schemes which only guarantee consistency but not a specific freshness.

Requirement (2) is important because most OLAP applications are susceptible to inconsistencies even though they do not mind stale data: For instance, think of a scenario where the sum of the sales in sales regions 'California North' and 'California South' is not equal to the one of 'California', i.e., the state as a whole. Clearly, it does not depend on the freshness of the data whether or not this is acceptable.

Our contribution is the design, implementation, and evaluation of a middleware that meets the requirements identified so far. Its core is a new protocol called FAS (*Freshness-Aware Scheduling*) with the following characteristics: The middleware directs all updates to a designated *OLTP node*. It routes a query that is part of a read transaction to one of the remaining cluster nodes, subsequently referred to as *OLAP nodes*. It propagates updates to OLAP nodes by deferred bulk refresh transactions. Refresh transactions start depending on the freshness of the cluster nodes. Our paper will investigate various alternatives to accomplish this. Further important questions are as follows: Are global correctness and scalability of OLAP throughput mutually exclusive? Can we effectively trade freshness of data for query performance? How expensive is access to up-to-date data using our middleware?

To answer such questions, we have fully implemented a prototype and have conducted an extensive experimental evaluation of FAS. It turns out that it has the following characteristics: our middleware does not become a bottleneck for OLAP even for large clusters of up to 128 nodes. Response times of clients which accept up to 20 minutes old (but nevertheless consistent) data are about 30% faster than those of clients asking for up-to-date data. Even if all clients ask for up-to-date data, query and update throughput is higher by more than an order of magnitude as compared to synchronous updating.

Finally, researchers recently have proposed data compression schemes and approximative query-evaluation techniques, e.g., [7], including techniques that allow to trade result quality for query-answering time. It is important to note that they are orthogonal to our current concern. We have consciously decided to keep these issues separate, in order to carry out a 'noise-free', quantitative assessment of our middleware. This is why physical design on all cluster nodes is identical. For the same reason, this study does not deal with intra-transaction parallelism. Of course, combining these techniques is natural and will lead to even better performance than the one from this study that is already pleasingly good.

This paper is organized as follows: Section 2 gives an overview of the system architecture. The next section presents freshness-aware scheduling in more detail. Section 4 reports on an extensive quantitative evaluation of FAS. Section 5 reviews related work. Section 6 concludes.

2 System Model and Architecture

Types of transactions. With regard to transactions submitted by clients, i.e., *client transactions*, we distinguish between *read-only (OLAP) transactions* and *update transactions*. A read-only transaction only consists of queries. An update transaction comprises at least one insert, delete, or update statement – in the following shortly referred to as updates – next to arbitrarily many further SQL statements. In case of a read-only transaction, the client specifies a *freshness limit* for the data accessed. Furthermore, decoupled *refresh transactions* propagate updates through the cluster.

Architecture. A *cluster of databases* is a cluster of commodity PCs, each running an off-the-shelf commercial database system (DBMS) as transactional storage layer. We also refer to a database at the cluster nodes as a *component DBMS*. This paper assumes that all cluster nodes are homogeneous, i.e., they run the same DBMS and have the same logical database design. We distinguish between a dedicated *OLTP node* and n *OLAP nodes*. There is a *coordination middleware*, also referred to as *coordinator*, that administers the cluster. It is responsible for scheduling, routing, and logging of the incoming requests. While the cluster consists of off-the-shelf hardware and software components, we have implemented the cluster coordinator ourselves. It comprises an *input queue*, a *scheduler*, a *router*, a *refresher*, and a *logger* (cf. Figure 1). Clients submit read-only and update transactions to the middleware. The OLTP node serves as a primary node where all updates will first be executed. Queries arrive at an *input queue*. The input queue is not processed in a 'first-in-first-out' manner. Instead, the *scheduler* decides in which order to process the incoming requests (a waiting time limit avoids starvation). There typically are several

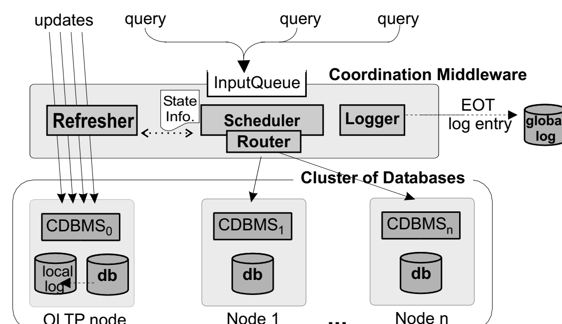


Figure 1: System Architecture.

OLAP nodes where a query of a read transaction may execute. The *router* chooses one of these nodes for each query. To do so, the coordination middleware maintains some global *system state* information, e.g., the freshness of each node.

Transaction management by the coordination middleware guarantees global correctness and consistency. FAS deploys a two-layered *open-nested transaction model* [23]: The queries of read-only transactions as submitted by clients are executed as separate subtransactions in the component DBMSs (cf. Section 3.3). The coordination middleware also contains a global *logger*. It keeps track of the update transactions on the OLTP node and their decoupled refresh transactions on the OLAP nodes. The latter are controlled by the *refresher*. This allows to be globally correct without distributed commit processing as with, e.g., two-phase-commit (2PC).

3 Freshness-Aware Scheduling

Freshness-aware scheduling (FAS) comprises replication management and mechanisms of multiversion concurrency control. The notion of freshness of data is crucial in the context of FAS. We will presently discuss freshness metrics, before introducing FAS in more detail. Subsection 3.3 describes how to apply the concepts of FAS to our architecture, while Subsection 3.4 presents the actual FAS protocol. Finally, Subsection 3.5 discusses refreshment strategies.

3.1 Freshness of Data

Freshness measures are closely related to the notion of coherency [2, 9, 8]. With our approach, a so-called *freshness index* $f(d) \in [0, 1]$ measures the freshness of some data d. This freshness index reflects how much the data has deviated from the up-to-date version. Intuitively, a freshness index of 1 means that the data is up-to-date, while an index of 0 tells us that the data is "infinitely" outdated. There are several freshness metrics possible [20]. *Delay Freshness* is workload-independent and is the most intuitive alternative of specifying the freshness needs of a client.

Delay freshness reflects how late a certain cluster node is as compared to the up-to-date OLTP node. It is based on the period of time between the last propagated update and the most recent update on the up-to-date node. Let $\tau(c)$ denote the commit time of the last propagated update transaction on an OLAP node c, and $\tau(c_0)$ the commit time of the most recent update transaction on the OLTP node. Then the delay freshness index is defined as $f(c) = \frac{\tau(c)}{\tau(c_0)}$.

With our implementation of FAS, the freshness index is computed at the level of entire databases. In principle, this granularity could also be finer, e.g., on the level of relations. We are however interested in the performance characteristics of freshness-aware scheduling with many nodes, e.g., more than a hundred. With such numbers, the finer granularity on the, say, level of relations would require considerably more effort from the scheduler. In particular, this would increase the overhead of both bookkeeping and the propagation of updates. Hence, we do not follow up on such alternatives in this current study, and leave this for future work.

3.2 Overview of Freshness-Aware Scheduling

FAS interleaves the execution of read-only transactions and update transactions. This will improve the freshness of the data accessed by queries. They may also access up-to-date data, if requested. However, we do not want to sacrifice correctness — the protocol shall guarantee one-copy serializability [3].

With FAS, individual queries access just one cluster node. However, the router can send each query of a read-only transaction to a different OLAP node in order to improve performance [22].

Replication and Correctness. A naïve approach to global correctness would use synchronous replication where each update immediately goes to all replicas, also referred to as eager replication. However, such approaches do not scale with the cluster size [10, 4]. Hence, FAS propagates updates asynchronously. It follows a primary-copy replication scheme with deferred refreshment.

The coordination middleware executes updates first on the OLTP node. The number of update transactions that run in parallel on the OLTP node is not restricted. After an update transaction finishes, as soon as a refresh is activated (cf. Section 3.6), the refresher propagates the changes to the remaining replicas using decoupled refresh transactions. In more detail, each of them refreshes one node and is activated separately. Each node guarantees locally serializable executions. In addition, FAS ensures *read consistency*: it propagates refresh transactions in a way that query-only transactions always see the same version during their lifetime. This has to be handled with care because with FAS, the router can send each query of a read-only transaction to a different OLAP node. Routing of queries of the same transaction to different cluster nodes is beneficial because of caching effects [22].

Freshness as Quality-of-Service. Next to guaranteeing serializability, freshness-aware scheduling aims at improving query response time. The idea is to introduce freshness of data as new Quality-of-Service parameter for transaction processing. This should allow to explicitly trade freshness of data accessed for query performance. The freshness limit is an additional constraint for query routing. Only

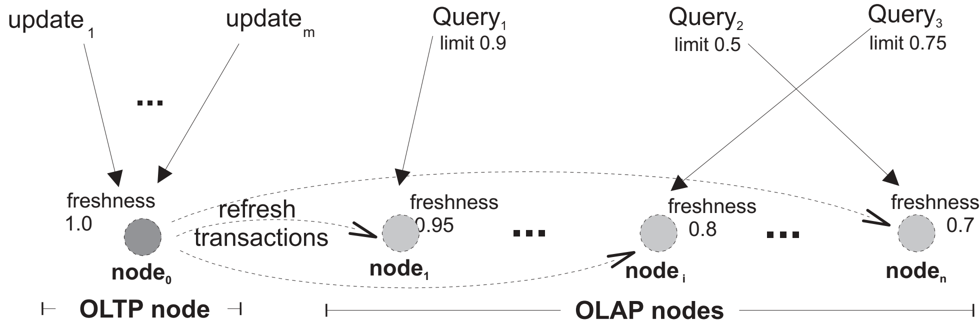


Figure 2: Principle of freshness-aware scheduling.

cluster nodes with a freshness above the given lower bound will be considered during query routing. Consequently, the higher the specified minimum freshness is, the smaller is the portion of the cluster to which the corresponding query may be routed. In the worst case, no node is available with the requested degree of freshness, and the coordination middleware must activate update propagation first. Hence, although FAS follows a lazy primary-copy replication approach with deferred updates, it nevertheless allows queries to access the most recent data.

Example 1. Figure 2 shows three queries with different freshness limits. The first query asks for data with a degree of freshness of at least 0.9. Only the first node has a freshness index that meets this limit and hence is the only possible target node. In contrast, Query 2 is asking for data with a freshness of at least 0.5. This freshness limit is met by all cluster nodes. Hence, FAS is free to route Query 2 to any of the OLAP nodes. In Figure 2, the last node is chosen to serve the query. ■

Further Performance Issues. The coordination middleware prohibits read transactions on a node while a refresh transaction executes. This is not a limitation: Previous research has shown that OLAP queries should be executed with a multiprogramming level of one per node in order to avoid obstruction effects [21] (this of course does not apply to the degree of multiprogramming at the OLTP node). Furthermore, even though a refresh transaction temporarily blocks one node, query evaluation still takes place in parallel on the remaining OLAP nodes. This is because it is unlikely that refreshes of all nodes of a large cluster happen at the same time.

Another important optimization is to pool several updates into a *bulk refresh transaction* in order to minimize the slowdown of query performance. It executes much faster than individual refresh transactions [12]. The reason is that the system does commit processing only once (with the same effect as group commit). However, for ease of presentation, we will leave this last optimization aside until Section 3.4.

3.3 Guaranteeing Correctness and Freshness

FAS has to guarantee that (1) refresh transactions are executed on all OLAP nodes in the same serialization order as the original update transactions on the OLTP node, and (2) all queries of a read transaction access data with the same freshness. To show formally that FAS meets these requirements, we rely on a two-layered *open-nested transaction model* [23]: Each global client transaction t consists of a set of SQL actions, which are executed as separate (flat) subtransactions in one component DBMS. Refresh transactions are subtransactions of a (global) update transaction. The benefit is that update and refresh transactions execute and commit independently, while still ensuring transactional guarantees from the perspective of the client. FAS keeps track of the propagation of updates, i.e., which refresh transactions are already done and which ones remain to be activated.

Figure 3 illustrates this. A global update transaction t is submitted by a client. First, its subtransaction t_u is executed on the the OLTP node c_0. After commitment of this first subtransaction, decoupled refresh transactions $r_{t,c_1}, \ldots, r_{t,c_n}$ propagate the updates to all replicas. They typically start at different times (cf. Section 3.6), so that the cluster as a whole is not blocked. Read-only transactions are also decomposed into subtransactions and routed to one or more OLAP nodes.

This layered approach helps to define what 'up-to-dateness' means, and to ensure that clients see data of the specified degree of freshness. FAS defines 'up-to-dateness', i.e., freshness 1, as the state of the OLTP node at the point in time when the first query of a read transaction is started. When an OLAP transaction wants freshness 1, FAS first propagates all updates committed before this point in time to the target node. FAS ensures that this is done in the serialization order of the OLTP node. This order is known to the coordination middleware if c_0 provides commit-order serializability [5]. If this is not the case[1], the coordination middleware must enforce a certain seri-

[1]For example, Oracle uses a protocol called *snapshot isolation* which does not provide commit-order serializability.

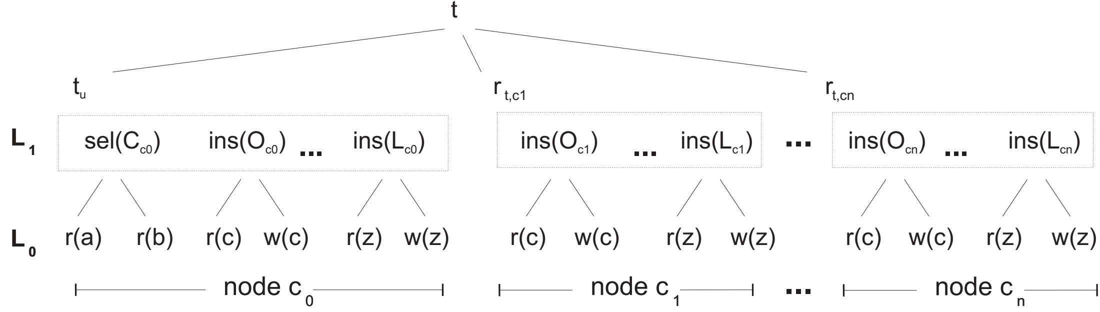


Figure 3: Conceptual layered transactions with FAS.

alization order, e.g., by using a ticket mechanism or a locking protocol.

Next to the refreshment of nodes with transactional guarantees, FAS encompasses freshness-based routing of queries. In extension of traditional transaction models, FAS allows clients to specify a *freshness limit* f_t per read-only transaction for the data accessed. This means that transactions need not access the most up-to-date version of the warehouse but a version which meets the freshness limit. The implications to correctness are that queries accessing stale data are serialized *before* update transactions which have already committed but have not been propagated to the query's target node so far.

This requires that refresh and query transactions are correctly serialized on the various nodes. In particular, FAS must ensure that all queries of a read-only transaction are accessing OLAP nodes with the same degree of freshness. This is done similarly to multiversion concurrency control [3]. FAS maps the actions, i.e., the individual queries of a read-only transaction, onto actions on some specific replica version. Note that the freshness of target node implicitly defines the version of the accessed data item. The reason is that each individual cluster node is just a one-version component, even though the cluster is a multi-version system.

3.4 FAS Protocol

In the following, we present the actual FAS protocol.

Timestamps. Let C be a cluster of databases, $C = \{c_0, c_1, \ldots, c_n\}$, with c_0 being the OLTP node, and let f_t be the freshness limit of a read-only transaction. In order to implement the delay freshness metric, transactions are assigned a unique timestamp, denoted by $\tau(t)$. Furthermore, FAS assigns timestamps to cluster nodes, denoted by $\tau(c_i)$. The timestamp of the OLTP node is the commit time of the latest update transaction. The timestamp of an OLAP node is set by each bulk refresh transaction. It sets the timestamp to the latest timestamp of the update transactions whose updates it propagates. Finally, the scheduler assigns to a read transaction the freshness of the node accessed by its first query, denoted by ff_t, in order to ensure read consistency (cf. Algorithm 3.1, line 20).

Scheduling and Routing. Freshness-aware scheduling serializes read and refresh transactions on the OLAP cluster using a multiversion timestamp ordering approach: FAS transforms each action a_i from the input queue into a subtransaction on some specific versioned replica, based on the timestamp and freshness limit of the transaction. In more detail, it works as follows:

```
foreach a_i in queue loop                                          1
                                                                   2
  // initial set of candidate nodes?                               3
  if a_i is the first of t_i then                                  4
      candidates:={c|c ∈ C, c ≠ c_0 : f(c) >= f_{t_i}};            5
      // if candidates = ∅ ⇒ activate refresh                      6
  else                                                             7
      candidates:={c|c ∈ C, c ≠ c_0 : f(c) = ff_{t_i}};            8
      if candidates = ∅ then Abort(t_i) end if                     9
  end if                                                          10
                                                                  11
  // nodes not currently refreshed?                               12
  choices := { c | c ∈ candidates                                 13
                 : node c not refreshing };                       14
                                                                  15
  if ( choices ≠ ∅ ) then                                         16
      //router chooses target node                                17
      c_target := Router.Choose(choices,a_i);                     18
      // remember accessed freshness                              19
      ff_{t_i} := f( c_target )                                   20
      // start subtransaction                                     21
      StartSubtransaction(a_i, c_target);                         22
      RemoveEntry(queque, a_i);                                   23
  else if waiting_time(a_i) > limit then                          24
      // activate refresh of oldest node                          25
      // and start a_i on refreshed node                          26
  end if                                                          27
                                                                  28
  // an action remains in the input queue                         29
  // until a suitable target is available                         30
                                                                  31
end loop                                                          32
 // ∃ c_i : c_i is idle ⇒ activate refresh                        33
```

Algorithm 3.1: FAS Protocol.

The scheduler iterates through the input queue and checks for each action if possible target nodes are available. This means that the nodes must meet the transactions freshness limit (cf. lines 4–10) and must not be currently offline due to an active refresh (cf. lines 13 and 14). The actual choice of the target node is left to the router (cf. line 18). The objective of the router is to identify the node where the given query will evaluate fastest. [22] has already described suitable routing algorithms, hence we do not deal with the details of the router in this paper.

Finally, the current action a_i is started as subtransaction on the target node chosen (line 22) and removed from the input queue (line 23). It may happen that the scheduler cannot ensure read consistency because all nodes have been refreshed already. If this is the case, the corresponding read transaction is aborted (line 9). However, a straightforward extension of FAS that does some bookkeeping which versions are still needed could avoid this.

Update Propagation. If needed, the scheduler activates update propagation (cf. lines 6, 25, and 33). We consider different variants of update propagation (cf. Section 3.6). All of them refresh each OLAP node separately, and they refresh the oldest node, i.e., the node with the lowest freshness index. The refresher that is part of the coordination middleware executes all committed updates which have not been propagated to the this node as one *bulk refresh transaction*. This is an important performance optimization, as compared to executing separate refresh transaction. Bulk refresh transactions are an extension of the layered transaction model as discussed in Section 3.3. However, as the refresher composes them in the serialization order of c_0, correctness is still guaranteed.

Ensuring atomicity and recovery naturally imposes a certain overhead, quantified in [4]. Global recovery cannot be shifted to the component DBMS, but the coordination middleware must provide it. In order to ensure atomicity of update transactions, the logger at the middleware layer (cf. Figure 1) performs bookkeeping of the refresh transactions. In addition, the OLTP node keeps a local log of committed updates. We implemented this local log using database triggers which add entries to a log table for each successful update. The coordination middleware can derive from both local and global log the state of update propagation throughout the cluster. This allows to do a forward recovery in case of a failure of the coordinator by continuing with missing refresh transactions. A further performance optimization is that the refresher caches all committed but not completely propagated updates in main memory to avoid reading the log of the OLTP node for each refresh.

3.5 Scalability and Availability

An important issue is to ensure that our coordination middleware does not become a bottleneck even for large cluster sizes. The focus of this current work is to provide the functionality of FAS, i.e., global correctness and freshness guarantees, and to experimentally explore its performance characteristics for a large cluster (128 nodes). In this study, several parts of the coordination middleware are centralized. For example, the scheduler and its state information cannot be distributed as it needs to do bookkeeping, e.g., of the degrees of freshness of the cluster nodes. There is also a central input queue in order to allow for query routing.

Other components of the coordination middleware can be distributed more easily. For example, scheduler, refresher, and logger do not have to run on the same node. The refresher can monitor committed update transactions on the OLTP node independently of the scheduler. It is activated by the scheduler just when needed. It then executes a bulk refresh transaction in parallel to the ongoing scheduling of queries on non-refreshing OLAP nodes. The global logger just keeps track of the successful update propagation, which is again widely independent of scheduling and routing. Our implementation of FAS further reduces synchronization of these middleware components by using multithreading. The evaluation will show that our coordination middleware indeed scales with increasing OLAP workloads for the considered cluster sizes of up to 128 nodes.

Availability has not been a topic of this current work. However, fail-safety can be easily increased by deploying hot-standby techniques for the machines running the coordination middleware. As FAS makes heavy use of replication, fail-safety at the cluster level itself is already very high.

3.6 Refreshment Strategies for FAS

So far, we have addressed the questions *how* scheduling can be accomplished. In this section, we concentrate on performance aspects of freshness-aware scheduling, e.g., *when* the refreshment of cluster nodes starts. We discuss different variants for update propagation which should have an effect on performance. The general intention is to hinder queries as little as possible. This can be achieved by varying the frequency of update propagation. This depends on the strategy when the scheduler starts deferred refresh transactions. We study three basic variants: *asap*, *on-demand*, and 1-*idle*.

asap The standard approach to update propagation is to propagate changes as soon as possible [17]. With such a refreshment strategy, cluster nodes will be refreshed between queries. In other

words, whenever a query finishes and updates have occurred in the meanwhile, the execution of a refresh transaction is started on the corresponding cluster node. The freshness index is not used at all. *asap* will serve as a reference point for the evaluation.

on-demand defers refreshment until there is a read transaction with a freshness limit that no node can meet. This is the case if its freshness limit is higher than the maximum of the freshness indices of the OLAP nodes. A refresh transaction will then start on the cluster node with the lowest freshness index.

m-idle defers refresh transactions until m cluster nodes have become too old so that they cannot fulfill the freshness requirements of any query in the input queue and hence become idle. The actual number of nodes which have become too old is a parameter of this strategy. However, it turned out that *1-idle* performs best in most cases [20], and we will not look at the case of $m > 1$ in the following.

An *asap* activation of refresh transactions is the state-of-the-art method for lazy replication [17]. The intention is to propagate updates through the cluster as fast as possible. In contrast, the two other approaches defer update propagation as long as queries are still satisfied with the freshness of data provided by the cluster. They differ in the criterion when to refresh a cluster node. With *on-demand*, refreshment may only be activated by a read transaction with a freshness limit above the freshness indexes of all OLAP nodes (cf. Algorithm 3.1, line 6). *1-idle* defers the activation even longer until one node has become so old that it cannot be used even for the read transactions with the lowest freshness limit — and hence becomes idle even if there are queries waiting in the input queue (cf. Algorithm 3.1, line 33). However, FAS ensures that no query is waiting longer than a given time limit (cf. line 24). This avoids starvation.

Example 2. Consider a cluster consisting of two nodes, c_1 and c_2. c_1 has freshness 0.85, and c_2 has freshness 0.7. The input queue of the coordinator contains three read transactions t_1, t_2, and t_3 with freshness limits 0.8, 0.9, and 0.6, respectively. *on-demand* will start t_1 at node c_1, but leave t_2 and t_3 waiting in the input queue. The reason is that t_2 is asking for a degree of freshness not provided by the cluster. *on-demand* therefore takes node c_2 offline in order to refresh it first, even though t_3 could be served. With *1-idle*, the cluster nodes serve t_1 and t_3, as the freshness of c_2 still meets the limit of t_3. ■

1-idle prefers queries with low freshness limits. This is because it waits with update propagation until even those queries cannot use any node any more. In contrast, one might expect that the *on-demand* strategy leads to lower response times than *1-idle* for queries asking for more up-to-date data. However, the evaluation will show that the opposite is true.

4 Evaluation

In the following, we are interested in the performance characteristics of FAS. We have implemented a full-fledged prototype, i.e., the coordination middleware, and have conducted an extensive experimental evaluation using our prototype on a cluster of 128 nodes. We have started by comparing different refreshment strategies. Based on the results, we have further evaluated the overhead and scalability of freshness aware scheduling. Finally, we have also compared FAS to synchronous updates.

4.1 Evaluation Setup

The prototype comprises the cluster of databases, a designated OLTP node, the coordinator, and a client simulator. The evaluation has been conducted on a cluster of databases consisting of 128 PCs (1 GHz Pentium III, 256 MBytes RAM, and two SCSI harddisks) each running Microsoft SQL Server 2000 under Windows 2000 Advanced Server. We generated the databases according to the TPC-R benchmark with a scaling factor 1 (with indexes about 2 GBytes). The OLTP node, the global log, and the client simulator are Pentium II 400 MHz machines with the same software configuration. The coordinator runs on a separate PC with two 1GHz Pentium III and 512 MBytes RAM. All nodes are interconnected by a switched 100 MBit Ethernet.

4.2 Influence of Refreshment Activation

First, we are interested in how the refreshment strategies influence update and query performance and actually, which refreshment strategy outperforms the others. Hence, we are varying the refreshment activation strategy.

In order to compare these three alternatives for lazy replication refreshment, we used a workload of 10 concurrent update streams and 64 concurrent query streams, denoted as $(10, 64)$. The updates corresponded to the TPC-R refresh function RF1, while the query streams encompassed randomized TPC-R queries. In this series of experiments, the cluster consisted of 32 nodes plus one dedicated OLTP node with a multiprogramming level of ten. We measured the performance of the different refresh activation methods with different mean freshness limits of the query streams. The clients used mean freshness limits between 0.6 and 1. A decrease of the freshness

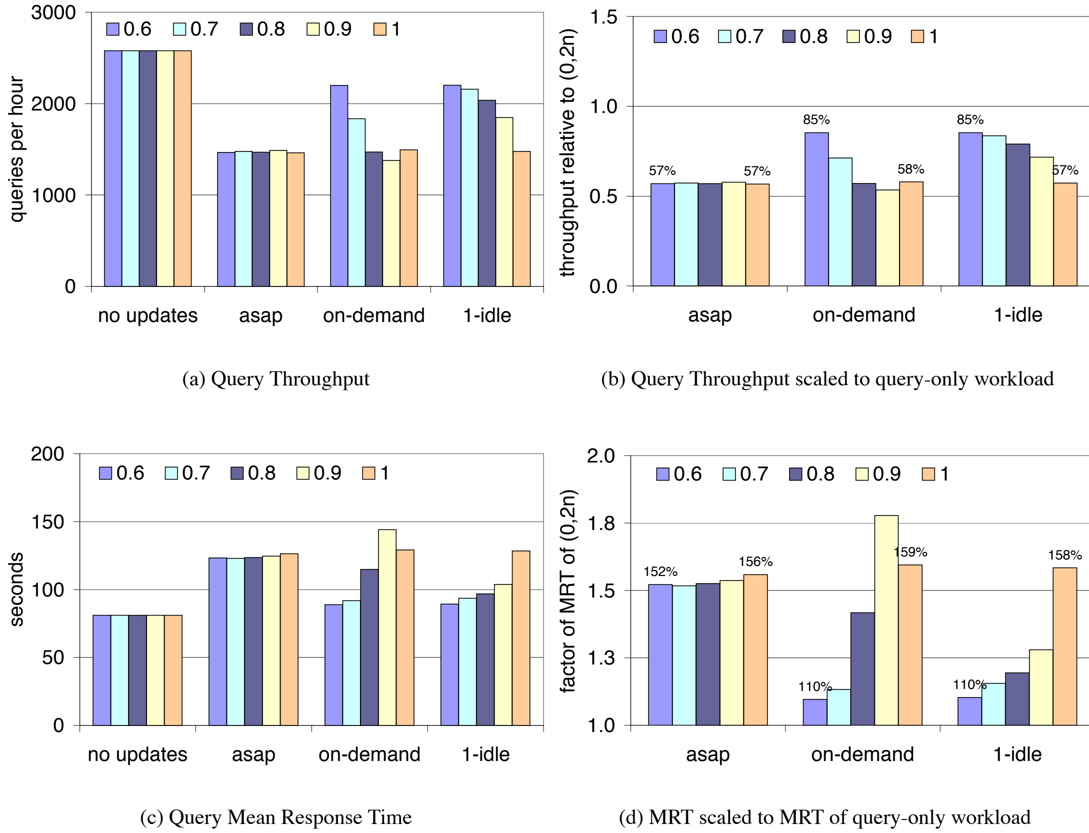


Figure 4: Influence of refreshment policy on performance of freshness-aware scheduling.

index of 0.1 corresponded to a maximal allowed refreshment delay of 5 minutes. The length of the observation period was 20 minutes.

Figure 4 contains the respective figures. It shows that the *asap* activation approach leads to mean response times around 50% higher than without any updates. Throughput also decreases to about 60% of the one without updates. Both ”lazy” approaches to refreshment activation, *1-idle* and *on-demand*, yield a much better performance. For queries with a low freshness limit like 0.6 in particular, they lead to much improved mean response times and throughputs. They reach about 85% of the original throughput without updates. Especially *1-idle* slows down queries with a low freshness limit only slightly. Figure 4(d) also shows that our original idea to trade up-to-dateness of the accessed data for query performance actually works: With *1-idle*, querying data with a freshness limit of 0.6 has about 50% less slowdown than asking for up-to-date data.

An unexpected effect is that *on-demand* yields a slightly worse performance than *1-idle*, especially with mean freshness limits between 0.8 and 0.9. For example, clients asking for data with a minimum freshness of 0.9 with *on-demand* end up with a longer response time as if they would ask for up-to-date data! This does not happen with *1-idle*.

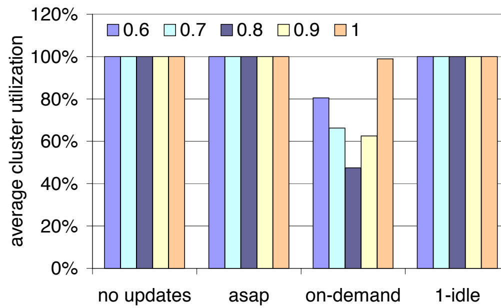
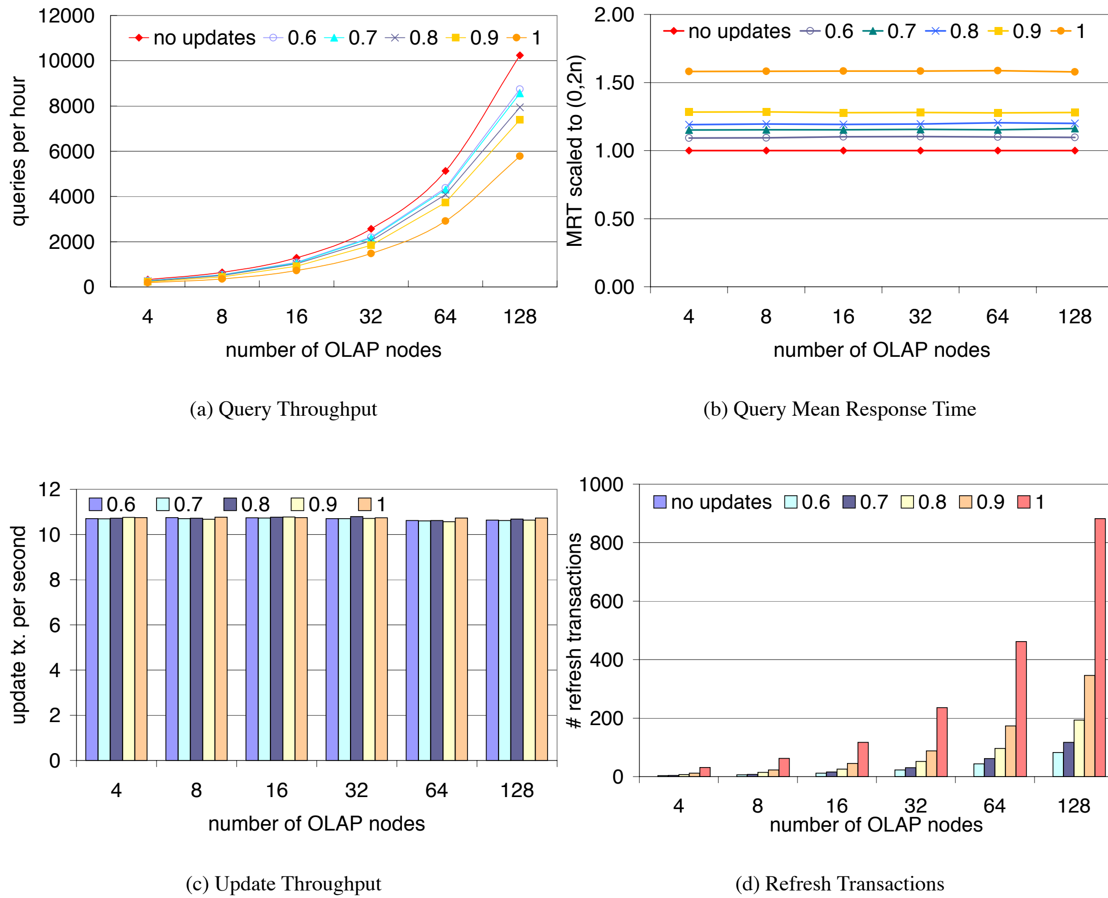


Figure 5: Cluster usage of refreshment policies.

The reason is that with *on-demand* activation the coordinator is satisfied if there is at least one node of the cluster which is fresh enough to answer an incoming query. Consequently, read transactions with high freshness limits often can use only a small ”fresh” subset of the actual cluster while queries with smaller limits have more choices. This means that the higher the mean freshness limit, the more queries are competing for a small number of fresh cluster nodes. Figure 5 illustrates the mean cluster utilization with the



(a) Query Throughput

(b) Query Mean Response Time

(c) Update Throughput

(d) Refresh Transactions

Figure 6: Influence of freshness requirement on query and update performance.

different methods. With the *1-idle* strategy, it is always 100%, while with *on-demand* more and more cluster nodes remain unused. This effect does not show when all clients ask for freshness 1, so that every transaction activates a refresh. The effect also does, by definition, not appear with *1-idle* activation, which takes the converse approach: Here, it will not happen that any node remains idle. This would mean that the node has become too old for all active transactions in the system, and hence it is refreshed.

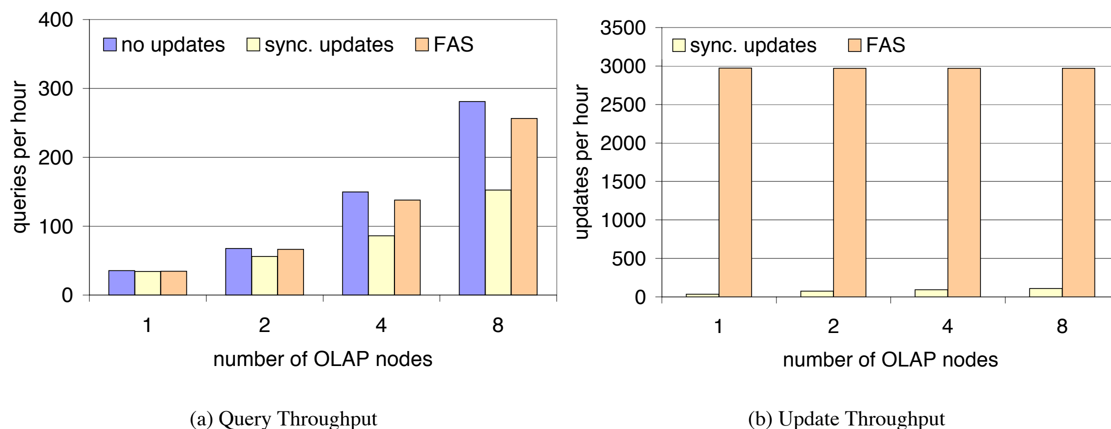
4.3 Influence of the Freshness Parameter

Let us further investigate the influence of the freshness parameter on performance. In the following, FAS always deploys *1-idle* refreshment activation. We have used a dynamic workload of $(10, 2n)$ for these experiments, i.e., ten update streams concurrently executed with twice as many querying clients as there are nodes in the cluster (n = size of cluster).

We again varied the mean freshness requested by read transactions from 0.6 up to 1. The results are shown in Figure 6. We see that the slowdown of queries by the concurrent update stream for *1-idle* is around 10% up to 60% with regard to mean response time as compared to the no-update case $(0, 2n)$. If clients issue queries with mean freshness limit 0.6 (which means at most 20 minutes old), they obtain the results about 30% faster, compared to a requested freshness of 1. This is exactly the effect freshness-aware scheduling is targeting on: trading data 'up-to-dateness' for query performance.

The results also nicely show that there is no slowdown with increasing cluster size, as it would be the case with synchronous updates: we were doubling the number of clients and the cluster size at the same time, and mean response time did not change, but query throughput doubled. This means that at least up to 128 nodes, freshness-aware scheduling scales linearly with increasing cluster size.

The result is the same with regard to update performance. Figure 6(c) illustrates this. The throughput achieved by ten concurrent update streams remains constant, even with a large OLAP cluster of up to 128 nodes. Obviously, the coordination middleware can keep up with the updaters and the increasing OLAP workload (on 128 nodes, 256 query



(a) Query Throughput

(b) Update Throughput

Figure 7: Performance comparison of FAS and synchronous updates.

streams are active) without a slowdown for either queries or updates. At the same time, the CPU load of the scheduler with 128 nodes was only 30% in average. All this is the result of our efforts to avoid a bottleneck (cf. Section 3.5).

Figure 6(d) on the preceding page illustrates the reason for the performance gains of lower freshness limits. It shows how many refresh transactions are started on some OLAP node for the different mean freshness limits requested. Clearly, the larger the cluster or the higher the mean freshness requested, the more often updates are propagated throughout the cluster. Conversely, the smaller the freshness limit requested by clients the smaller is the overhead for refreshment.

4.4 Comparison with Synchronous Updates

In the following, we are interested in a comparison of FAS with a standard approach to consistent data with up-to-date guarantees. Although asynchronous replication approaches can ensure serializability, e.g. [1, 6], none of them can also guarantee access to up-to-date data. For this reason, we are comparing FAS with synchronous update propagation. A synchronous updating strategy executes updates on all replicas within the same transaction. Typically, a two phase commit protocol (2PC) guarantees atomicity.

To carry out this comparison, we deployed a $(1, n)$ workload. It comprises one updater and as many readers as cluster nodes. FAS used the *1-idle* refreshment activation strategy. In order to be comparable to synchronous updating, which keeps all replicas up-to-date, all clients were requesting freshness 1 in the case of FAS. We also used only up to eight OLAP nodes. The reason why we used a smaller cluster and workload than before will become clear when we look at the results.

FAS outperforms synchronous updates both with regard to query and update throughput. Synchronous updates slow down significantly with increasing cluster size. Figure 7 shows that on eight nodes only 55% of the query throughput as without updates is achieved. On the other hand, FAS still achieves 91% of the non-update throughput on eight nodes, even though the readers asked for a freshness of 1, i.e., up-to-date data! This is much faster than in the previous experiments, but the workload is also much smaller.

The differences are quite dramatic with regard to update performance. FAS allows for an update throughput on the OLTP node of about 3000 update transactions per hour, each transaction consisting of five insert statements on average. The update throughput stays stable over the cluster size. In contrast, with synchronous updates each update transaction must obtain exclusive locks on all cluster nodes. Consequently, queries significantly slow down updates. The result is an extremely small update throughput of around a hundred update transactions per hour. FAS is more than one magnitude better. — Summing up, this experiment illustrates the following: FAS shows the superior performance of lazy replication for this special case where each client requests freshness 1. Furthermore, it is transparent that FAS propagates updates asynchronously. Each client accesses up-to-date data, as in the case with synchronous updating.

5 Related Work

FAS provides OLAP clients with constant performance, one-copy serializability, and freshness guarantees. This even holds if clients access up-to-date data. The design of a protocol with such characteristics is not obvious. It is well-known that synchronous updates do not scale with the cluster size [10]. But although asynchronous replication management has been intensively studied in the past, only few ap-

proaches actually guarantee a correct, serializable execution [1, 6]. They place some restrictions on the data placement or commit processing. [1] presents a serializable epidemic replication protocol. It relies on a distributed atomic commit protocol. This rules out this approach for large clusters. [6] avoids this — but only for certain configurations of primary-copy replication.

At the first glance, FAS is a restricted case of [6]. For instance, the protocol in [6] does not rely on the assumption that there is a distinguished coordination layer. Our current solution in turn contains a distinguished coordinator, and we assume that all requests are submitted to it, instead of going directly to individual nodes. This of course requires that it does not become a bottleneck. Our evaluation based on a full implementation has shown that this is indeed not the case even for a large cluster of 128 nodes. Actually, FAS benefits from this coordination layer, and can be more flexible than [6]: First, FAS allows that queries of the same read transaction are routed to different cluster nodes as long as they are read consistent. Second, FAS incorporates freshness guarantees. The combination of scalability, correctness, and freshness-awareness as Quality-of-Service is novel to replication management.

Online Warehouse Update Algorithms. FAS also goes beyond previous approaches to warehouse maintenance [19, 12, 11] that have focused solely on improving performance of update propagation. They neither have made the notion of freshness of data explicit, nor do they consider global correctness. While the cluster gives rise to further natural optimizations like intra-query parallelism or the use of more than one OLTP node with the data partitioned, such issues are orthogonal to our current work.

Approximative Query Evaluation. Our current concern is the design and assessment of the middleware, apart from physical design issues at the component level. Approaches like [7] propose data compression techniques that allow to trade result quality for query evaluation time. Broadly speaking, such approaches have the same objective, i.e., allowing the user to waive accuracy of results in exchange for better performance. But they are orthogonal and complement each other very well: different cluster nodes could hold different compressed versions of the database. The coordination middleware could then take into account that more sophisticated compression schemes typically induce higher maintenance costs.

[15] combines cache staleness and approximative query evaluation. The approach allows users to supply a quantitative precision constraint along with each query. The system then evaluates the query on both locally cached data and the master copy data in order to meet the precision constraint. In contrast, FAS aims as trading result *up-to-dateness* for query performance, and furthermore guarantees clients to access a consistent view of the database.

Commercial Cluster Products. Cluster of PCs have become an attractive hardware platform for commercial database vendors, too. While *Oracle 9i Real Application Cluster* follows a variant of shared-disk approach [16], most products such as *IBM DB2 UDB EEE* or *Microsoft SQL Server 2000*[18, 13] favor a shared-nothing architecture. They concentrate on data partitioning as physical design and efficient distributed, parallel query evaluation, typically for OLTP only. However, these systems do not address the problem of online warehouse updates. For example, the available replication mechanisms exploit either standard 2PC or full asynchronous replication protocols without any freshness guarantees.

6 Conclusions

Most data warehouses nowadays offer a compromise between freshness of data and maintenance costs. We think that this is a restriction, and our idea has been to allow to explicitly trade freshness of data for query performance. But at the same time, we do not want to sacrifice correctness. The architecture investigated here is a cluster of OLAP-components, with a middleware layer on top. The degrees of freshness of the cluster nodes may vary. Clients submit a query together with an explicit *freshness limit* as a Quality-of-Service parameter.

Our contribution is the design, implementation, and evaluation of a coordination middleware that schedules and routes updates and queries to cluster nodes. Its core is a new protocol called FAS (*Freshness-Aware Scheduling*) with the following characteristics: (1) The requested freshness limit of queries is always met, and (2) data accessed within a transaction is consistent, independent of its freshness. In particular, FAS can efficiently serve clients asking for up-to-date data. FAS makes use of the different degrees of freshness of the OLAP nodes in order to serve such queries which agree to access stale data sooner than queries asking for the latest data.

In a quantitative evaluation using a full-fledged prototype, we have investigated various alternatives to cluster refreshment and have compared FAS to both synchronous and asynchronous update propagation strategies. We could show the following important points:

- Our middleware does not become a bottleneck even for large clusters; up to 128 nodes, it allows a linear scaleup with regard to query response times while at the same time does not slowdown OLTP.

- Freshness-aware scheduling effectively allows to trade freshness of data for query response time; for example, response times of clients with a freshness limit 0.6 (data at most 20 minutes old) are about 30% faster than those of clients asking for up-to-date data.

- Update throughput is faster by more than an order of magnitude than with state-of-the-art synchronous update propagation, even if all clients are served with up-to-date data.

These performance improvements are achieved only by a sophisticated interleaving of updates and queries. A natural extension would be to have cluster nodes with different physical design. But even without such extensions, our results are positive. We conclude that freshness-aware scheduling is a promising approach to cope with high workloads of OLAP queries with different freshness requirements.

Acknowledgements. This work is part of the *PowerDB* cluster project currently conducted by the database research group at ETH Zurich. This project is partially sponsored by Microsoft Research.

References

[1] D. Agrawal, A. El Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems, May 12-14, Tucson, Arizona*, 1997.

[2] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems (TODS)*, 15(3):359–384, 1990.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the coordination overhead of synchronous replica maintenance in a cluster of databases. In *Proceedings of the 6th Int. Euro-Par Conference, Aug. 29 - Sept. 01, Munich, Germany*, pages 435–444, 2000.

[5] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, September 1991.

[6] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proceedings of SIGMOD 1999, June 1–3, Philadephia, USA*, 1999.

[7] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *Proceedings of 26th VLDB Conference, September 10–14, Cairo, Egypt*, 2000.

[8] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, Dallas, Texas, USA*, pages 117–128, 2000.

[9] R. Gallersdörfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *Proceedings of 21th VLDB Conference, September 11-15, Zurich, Switzerland*, 1995.

[10] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD Conference, June 4–6, Montreal, Quebec, Canada*, pages 173–182, 1996.

[11] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proceedings of 26th VLDB Conference, September 10-14, Cairo, Egypt*, 2000.

[12] A. Labrinidis and N. Roussopoulos. Reduction of materialized view staleness using online updates. Technical report, University of Maryland, 1998.

[13] http://www.microsoft.com/sql/default.asp

[14] M. Oguchi and M. Kitsuregawa. Parallel data mining on ATM-connected PC cluster and optimization of its execution environments. In *Proceedings of the 15 IPDPS Workshops, Cancun, Mexico*, 2000.

[15] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the 26th VLDB Conference, September 10-14, 2000, Cairo*, 2000.

[16] http://otn.oracle.com/products/oracle9i/

[17] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proceedings of 25th VLDB Conference, September 7-10, Edinburgh, Scotland*, 1999.

[18] IBM White Paper. IBM DB2 universal database on IBM Netfinity and GigaNet cLan clusters. Technical report, IBM Corporation, April 1998.

[19] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proceedings of the 1997 ACM SIGMOD Conference, May 13-15, Tucson, Arizona*, pages 393–404, 1997.

[20] U. Röhm. *Online Analytical Processing in a Cluster of Databases*. PhD thesis, ETH No. 14591, 2002.

[21] U. Röhm, K. Böhm, and H.-J. Schek. OLAP query routing and physical design in a database cluster. In *Proceedings of the 6th EDBT Conference, March 27-31, Konstanz, Germany*, 2000.

[22] U. Röhm, K. Böhm, and H.-J. Schek. Cache-aware query routing in a cluster of databases. In *Proceedings of the 17th ICDE Conference, April 2-6, Heidelberg, Germany*, pages 641–650, 2001.

[23] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.