ETL Queues for Active Data Warehousing

Alexandros Karakasidis
Univ. of Ioannina
Ioannina, Hellas
alex@cs.uoi.gr

Panos Vassiliadis
Univ. of Ioannina
Ioannina, Hellas
pvassil@cs.uoi.gr

Evaggelia Pitoura
Univ. of Ioannina
Ioannina, Hellas
pitoura@cs.uoi.gr

ABSTRACT

Traditionally, the refreshment of data warehouses has been performed in an off-line fashion. Active Data Warehousing refers to a new trend where data warehouses are updated as frequently as possible, to accommodate the high demands of users for fresh data. In this paper, we propose a framework for the implementation of active data warehousing, with the following goals: (a) minimal changes in the software configuration of the source, (b) minimal overhead for the source due to the active nature of data propagation, (c) the possibility of smoothly regulating the overall configuration of the environment in a principled way. In our framework, we have implemented ETL activities over queue networks and employ queue theory for the prediction of the performance and the tuning of the operation of the overall refreshment process. Due to the performance overheads incurred, we explore different architectural choices for this task and discuss the issues that arise for each of them.

1. INTRODUCTION

The demand for fresh data in data warehouses has always been a strong desideratum from the part of the users. Traditionally, the refreshment of data warehouses has been performed in an off-line fashion. In such a data warehouse setting, data are extracted from the sources, transformed, cleaned and eventually loaded to the warehouse. This set of activities takes place during a loading window, usually during the night, to avoid overloading the source production systems with the extra workload of this workflow.

Still, users are pushing for higher levels of freshness. *Active Data Warehousing* refers to a new trend where data warehouses are updated as frequently as possible, due to the high demands of users for fresh data. The term is also encountered as 'real time warehousing' for that reason [22]. To give an example, we mention [3], where a case study for mobile network traffic data is discussed, involving around 30 data flows, 10 sources, and around 2TB of data, with 3 billion rows. The throughput of the (traditional) population system is 80M rows/hour, 100M rows/day, with a loading window of only 4 hours. The authors report that user requests indicated a need for data with freshness at most 2 hours.

This kind of request is technically challenging for various reasons. First, the source systems cannot be overloaded with the extra task of propagating data towards the warehouse. Second, it is not obvious how the active propagation of data can be implemented, especially in the presence of legacy production systems. The problem becomes worse since it is rather improbable that the software configuration of the source systems can be significantly modified to cope with the new task (both due to (a) the down-time for deployment and testing and (b) the cost to administrate, maintain and monitor the execution of the new environment).

So far, research has mostly dealt with the problem of maintaining the warehouse in its traditional setup [10, 15, 16, 18, 21]. Related literature presents tools and algorithms for the population of the warehouse in an off-line fashion. In a different line of research, data streams [1, 5, 17] could possibly appear as a potential solution. Nevertheless, at least until now, research in data streams has focused on topics concerning the front-end, such as on-the-fly computation of queries, without a systematic treatment of the issues raised at the back-end of a data warehouse. For example, to our knowledge, there is no work related to how streaming data are produced or extracted from data producers; not to mention the extra problems incurred when the data producers are operational systems.

To this end, in this paper we attempt to approach the problem from a clean sheet of paper. We investigate the case where the source of the warehouse is a legacy system. The specific problem involves the identification of a software architecture along with appropriate design guidelines for the implementation of active warehousing. We are motivated by the following *requirements* in achieving this goal.

1. *Maximum freshness of data*. We want to implement an active data warehousing environment to obtain as fresh data as possible in the warehouse.
2. *Smooth upgrade of the software at the source*. We wish to implement a framework where the modification of the software configuration at the source side is minimal.
3. *Minimal overhead of the source system*. It is imperative to impose the minimum additional workload to the source.
4. *Stable interface at the warehouse side*. It would be convenient if the warehouse would export a stable interface for its refreshment to all its source sites.

The grand view of our environmental setup is depicted in Figure 1. A set of *sources* comprise source data and possibly *source applications* that manage them (for the case of legacy sources) or DBMS's for the case of conventional environments. The changes that take place at the sources have to be propagated towards the warehouse. Due to reasons of semantic or structural incompatibilities, an *intermediate processing stage* has to take place, in order to transform and clean the data. We refer to this part of the system as the *Active Data Staging Area* (ADSA). Once ready for loading, the data from the intermediate layer are loaded at the warehouse, through a set of *on-line loaders*.

IQIS'05, June 17, 2005, Baltimore, MD, USA.

Mapping this grand view to concrete technical choices requires the tuning of several components of the architecture. Following, we quickly summarize our findings that affected our architectural choices.

Starting with the sources, in this paper, we have focused on legacy systems. Apart from the requirement of minimal changes at the source side, legacy sources pose the interesting problem of having an application (instead of a DBMS) managing the data. We modify a library of routines for the management of data to allow the interception of the calls without affecting the applications. The modification involves (a) inserting no more than 100 lines of code to a library of routines for source management and (b) recompiling the application (which was not affected), over this library. Also, as far as the communication between stages is concerned, we transmit blocks of records for reasons of performance and minimal overhead of the source system.

The internal architecture of the intermediate layer (ADSA) is not obvious, either. For each ETL activity, we employ a queue to store incoming records before they are processed. Each activity processes the incoming data on-line and then passes its output to the next queue for further processing. Again, for reasons of performance, the unit of exchange is blocks of records and not individual records. We do not assume a fixed set of ETL operators, but rather we provide a taxonomy of such operations, based on their operational semantics. New operators can be added to the taxonomy as they are defined. To predict the performance of the system, we employ queue theory for networks of queues (cf. section 2.1 for a reminder of queue theory). Our experimental results indicate that the assumption of an M/M/1 queue for each of the ETL activities provides a successful estimation.

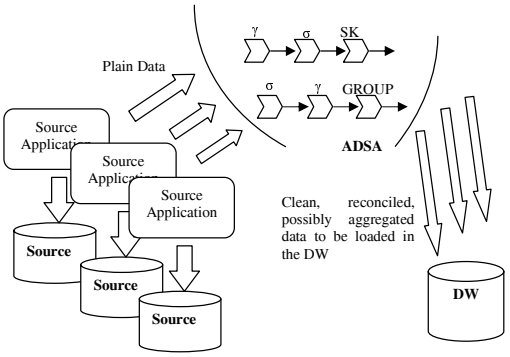

Fig. 1. Architecture Overview

To implement the requirement for stable interface at the side of the warehouse, the data are further propagated towards the warehouse through an interface involving web services [2]. The need for web services as the technical solution for populating the warehouse with fresh data is not self-evident and requires justification. In fact, web services are known to be rather heavy middleware in terms of resource consumption [9], which potentially jeopardizes the requirement of fresh data and minimal overhead. The main advantages of web services compared to other middleware solutions (RPC, ORB's, message queues, etc) are: (a) interoperability, meaning that they can be deployed in all platforms and configurations and (b) possibility of exporting them outside the intranet of an organization. We emphasize the interoperability property: in a large organization, there is a wide variety of data sources, involving several platforms and configurations. Web services are syntactically reliable, as they can provide a common, stable interface for the warehouse to all these sources without requiring major design and integration effort. Also, this loose coupling of sources and the warehouse results in minimal impact in the case of changes, either at the source or at the warehouse. Obviously, performance has been a concern too. Still, as we discuss in Section 4, our experiments indicate that the overall delay, incurred by the adaptation of a solution based on web services is rather small, especially if one is willing to trade resources (mainly main memory) for freshness.

In a nutshell, our contributions can be listed as follows:

- We set up the architectural framework and the issues that arise for the case of active data warehousing.
- We develop the theoretical framework for the problem, by employing queue theory for the prediction of the performance of the system. We provide a taxonomy for ETL tasks that allows treating them as black-box tasks, without the need of resorting to algebraic, white-box descriptions of their functionality. Then, standard queue theory techniques can be applied for the design of an ETL workflow.
- We provide technical solutions for the implementation of our reference architecture, achieving (a) minimal source overhead, (b) smooth evolution of the software configuration at the source side and (c) fine-tuning guidelines for the technical issues that appear.
- We validate our results through extensive experimentation. Our implementation suggests that our theoretical formulation successfully predicts the actual performance of the system.

The rest of this paper is organized as follows. In Section 2, we set up the problem theoretically and in Section 3, we present the different architectural choices and the technical challenges that each of them incurs. In Section 4, we present the experimental evaluation of the proposed framework. Finally, in Section 5, we present related work and in Section 6, we conclude with our results and present topics for future research.

2. QUEUE THEORY FOR ETL ACTIVITIES

In our architecture, data flows from the sources towards the warehouse, through an intermediate data processing stage. In this stage, data sustain various types of filtering and transformations. We employ queue theory as the cost model that predicts the data delay and the system overhead at this intermediate stage. We model each ETL activity as a queue in a queuing network. We provide a simple taxonomy for ETL activities, showing how to derive a simple queue model for them, without delving into their internal semantics. In this section, we start with some fundamentals of queue theory, then we move on to discuss a taxonomy of ETL operations and, finally, we conclude with the presentation of queue networks.

2.1. Preliminaries

Fundamentally, in a queuing model, a sequence of customers arrives at a server. If a customer arriving at the server finds the server occupied, it waits in the queue until its turn to be served comes. After the customer is served, it leaves the system [11]. If λ customers arrive at the system per time unit, then the mean inter-

arrival time is equal to $1/\lambda$. Similarly, if μ customers leave the system per time unit, then the mean service time is equal to $1/\mu$. Based on these parameters, we also define $\rho=\lambda/\mu$ as the traffic intensity which denotes the server utilization. We require that $\rho<1$ or the queue length can become unbounded.

The distribution of the arrival and the service rates can take different values (Poisson, constant, etc). Depending on these distributions, different equations hold for predicting the mean length of the queue and the mean service time for each customer. A full discussion of these properties falls outside the scope of this paper; therefore we refer the interested reader to [11, 23] for a detailed discussion.

A fundamental relation between the mean number of customers in the system N, the customer mean arrival rate in the system λ, and the mean time T that a customer remains in the system is given by Little's law. This relation is formulated as $N=\lambda*T$ and its importance resides in the fact that this equation holds for every type of queuing system irrespectively of the arrival and service rate distributions. By applying Markov Theory and Little's law to a queue with Poisson arrivals and exponentially distributed processing times, (also known as M/M/1 queue), we can estimate the mean response time of the system $W = 1/(\mu—\lambda)$ and the mean queue length $L=\rho/(1-\rho)$.

2.2. A Taxonomy of ETL Activities

Each ETL queue can direct customers to more than one subsequent queue, depending on the type of operation it performs. In queue theory, the composition of queues is treated by queue networks. The computation of the interesting properties of such networks depends on the nature of the involved individual queues. The question that arises is what kind of individual queues do the ETL activities produce. One possible way to answer this question is to define an extension of the relational algebra, specifically tailored for ETL purposes and study the properties of each operator from the viewpoint of queue theory. Since this would probably produce quite complex queues, we adopt a different, black-box approach and define a taxonomy of ETL transformations, based on the relationship of their input and output. This way, we practically categorize ETL tasks in families without delving in the particularities of their internal functionality. Specifically, the taxonomy of activities consists of the following categories: (a) Filters, (b) Transformers and (c) Binary Operations.

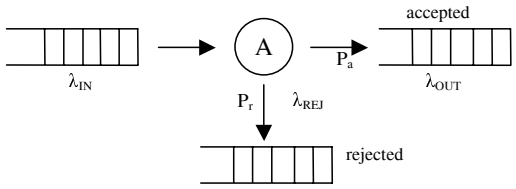

Fig. 2. Queuing model for multi-output activities

Filters examine each incoming tuple to determine whether it meets certain criteria. If these criteria are fulfilled, then a tuple is accepted and propagated towards an acceptance output. If not, it is rejected and possibly propagated towards a rejection output. We assume that tuple arrivals occur due to a Poisson process and service times follow an exponential distribution. We define the probability that some tuple i is accepted as P_a and the probability that some tuple i is rejected by the system as P_r. This is illustrated in Figure 2. It is obvious that $P_a+P_r=1$.

The filtering operations do not impose a change to the overall number of tuples making the following equation valid:

$$| \textit{tuples entering service} | = | \textit{tuples accepted} | + | \textit{tuples rejected} |$$

Also, these operations do not incur changes to the schema of the tuples entering the service facility compared to the schema of the exiting tuples. Typical operations of this category are not-null, domain and foreign key checks, selections, and in general, any type of operation, operating locally on a tuple and determining whether it will be further propagated or not. Due to their multiple outputs, filters can also act as routers for tuples whose destination depends on their value.

Considering the case of **Transformers**, tuples entering a transformer undergo changes to their value and/or their schema. We can distinguish two subclasses of Transformers taking into account the relationship between the number of tuples entering and the number of tuples exiting the transformation.

In the first case the two quantities are equal which means:

$$| \textit{tuples entering service} | = | \textit{tuples accepted} |$$

We assume that tuple arrivals occur due to a Poisson process and service times follow an exponential distribution, in other words we have the same case with filters transformations. Again, we define the probability that some tuple i is accepted as P_a and the probability that some tuple i is rejected by the system as P_r. Since all tuples are accepted, we have: $P_a=1$ and $P_r=0$ (Figure 3). Examples of such transformations are the surrogate key transformation, the usage of functions for the derivation of new values and, in general, any transformation that derives an output tuple solely on the basis of the value of a single input tuple.

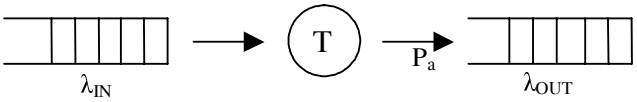

Fig. 3. Queuing model for single-output activities

In the second case, the number of tuples entering the system is different compared to the number of tuples exiting and in specific:

$$| \textit{tuples entering service} | > | \textit{tuples accepted} |$$

This occurs because some of the tuples entering service are aggregated or merged. We assume that tuple arrivals occur due to a Poisson process and service times follow an exponential distribution. The problem with this kind of transformations is that practically queue customers disappear and new customers are produced by each transformation. To model this property in terms of queue theory, we make the assumption that depending on the aggregation or merging factor, some of the incoming customers continue and some exit the system. In other words, we assume that some of the tuples, after being transformed, continue through the system as accepted. The number of these tuples equals the number of tuples produced as a result of the transformation. The rest of the tuples are assumed to be rejected by the system after their service and exit the system. The following equation holds:

$$| \textit{tuples rejected} | = | \textit{tuples entering service} | - | \textit{tuples accepted} |$$

Again, we define as P_a the probability that some tuple i is accepted and P_r the probability that some tuple i is rejected by the system: $P_a+P_r=1$. Given the aggregation factor of an incoming set of data, we can easily compute the acceptance and rejection rate as well as the respective routing probabilities. The routing probabilities are:

$$P_a = \frac{|result\_tuples|}{|input\_tuples|} \text{ and } P_r = \frac{|input\_tuples| - |result\_tuples|}{|input\_tuples|}$$

The third class of ETL activities deals with **Binary** operators. This is the case where data from multiple sources are combined and a single outgoing stream is produced. Examples of such operations involve variants of the join operation, including the join of data from different tables, as well as difference and update detection operations among different snapshots of the same table. [14] describe a window-based hash join algorithm for continuous streams. In the context of ETL, we make the following assumptions and observations:

- One of the two inputs is considered as the *primary input flow*. Tuples of this flow are checked over filters or transformed according to the values of some other relation and ultimately, either propagated towards the warehouse or rejected.
- The second input of the operator is acting as a *regulator of the primary flow*. In other words, its values are only needed in order to determine the processing and routing of the tuples of the primary flow. For all practical purposes where active ETL functionality is needed (update detection, difference, facts joined with dimension values), a static snapshot of the regulator flow can even be assumed.
- Adopting the model of [14], both inputs arrive at the same queue – they simply undergo processing with different distributions of processing times.

In principle, a binary operator has to be dealt with as a multi-class queuing system, with one class for each flow (input or output) – see Figure 4. We refer the interested reader to [14] for such a treatment. Still, based on the aforementioned assumptions, we can avoid modeling the system as a multi-class queue, and deal only with the primary flow of the operator. In the rest of the paper, we will consider single-class queues, the tuples of which either (a) continue in the system or (b) are ultimately rejected. An interesting observation here is that no matter how many different categories of tuples enter the node for service, the output tuples can be assumed to belong in one of the two aforementioned categories.

We consider Poisson arrivals and exponential service times. As stated earlier, the two routing classes are accepted with probability P_a and rejected with probability P_r and as before $P_a+ P_r=1$. This type of operations does not impose a change to the overall number of tuples existing making the following equation valid (Figure 4):

$$|\ tuples\ entering\ service\ | = |\ tuples\ accepted\ | + |\ tuples\ rejected\ |$$

However, differently from Filters, the schema of the tuples possibly changes.

We can generalize the three aforementioned classes, through a **Generic Model**, where a node consisting of a single server serves possibly more than one classes of customers. All customers arrive according to a Poisson process and are serviced with exponential service times. The general case is depicted in Figure 4.

In the general case we can assume that tuples belonging to one of the two different classes of customers, say c_i, after their ETL transformation at the node, leave the system with probability P_{ri} and continue in the queue network with probability P_{ai}. Concerning the number of tuples in the system the following equation is still valid:

$$|\ tuples\ entering\ service\ | = |\ tuples\ accepted\ | + |\ tuples\ rejected\ |$$

Concerning the schema of the tuples before and after service, we observe that the schema changes in the general case, apart from the case of filters.

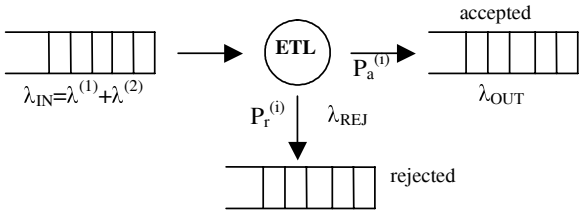

Fig. 4. Generic Model for ETL Queues

In the rest of this paper, we will follow the assumption of a primary input flow. This obviously results in forming an M/M/1 queuing node as the constructing element of our ETL queue network.

2.3. Queue Networks for ETL queues

Many queuing systems consist of a network of queues. In a *queuing network* (QN), a customer finishing service in a service facility is either immediately proceeding to another service facility or leaves the system. For our purposes, we assume that each node of this network consists of a single server with exponential arrival and exponential service times. One basic classification of queuing networks is the distinction between open and closed queuing networks. In an open network, new customers may arrive from outside (coming from a conceptually infinite population) and later on leave the system. In a closed queuing network, the number of customers is fixed and no customer enters or leaves the system. In our case, we are exclusively interested in open networks.

If an open queuing network is in steady state (i.e., the number of customers in the queue has converged over time), then for each node i, its arrival rate λ_i equals its departure rate μ_i. The arrival rate λ_i to node i is clearly the sum of all arrivals to i (including i itself). Assuming that i has N neighbors, the rate of external arrivals is λ_{0i}, and the probability of an arrival from its j-th neighbor is $p_{j,i}$, we have: $\lambda_i = \lambda_{0i} + \sum_{j=1}^{N} p_{j,i}\lambda_j$

These equations are called traffic equations and they can be transformed into a set of N simultaneous linear equations with a unique solution for M/M/1 nodes. In order to calculate the performance measures in queuing networks the steady state probabilities $\pi(k_1,\ldots, k_N)$ have to be found. The term $\pi(k_1,\ldots, k_N)$ denotes the probability of k_1 customers in queue 1, k_2 customers in queue 2 and so on. To this end, we employ Jackson's theorem that allows the calculation of the steady state probabilities of the whole network by separately calculating the probabilities of each

node, under reasonable assumptions (that our ETL queues fulfill) [23].

Jackson's Theorem [23]. If in an open network the condition $\lambda_i < \mu_i \cdot m_i$ holds for every $i \in \{1, ..,N\}$ (with m_i standing for the number of servers at node i) then the steady state probability of the network can be expressed as the product of the state probabilities of the individual nodes:

$$\pi(k_1,..., k_N) = \pi_1(k_1)\pi_2(k_2)... \pi_N(k_N)$$

Therefore, we can solve this class of networks in four steps:

1. Solve the traffic equations to find λ_i for each queuing node i.
2. Determine separately for each queuing system i its steady-state probabilities $\pi_i(k_i)$.
3. Determine the global steady-state probabilities $\pi(k_1,..., k_N)$. Derive the desired global performance measures.
4. From step 1, we can derive the mean delay and queue length for each node.

Methodology. How can we exploit the aforementioned theoretical analysis for designing ETL workflows for active data warehousing? *The design problem for active data warehousing involves predicting the mean delay and the queue length of ETL queues in the ADSA, given the source production rates and the processing power of the ADSA and the DW.*

The methodology for this task is straightforward. First, we classify each ETL task that we need to perform in one of the categories of our taxonomy. Then, we construct a queue network of such ETL queues. Finally, we solve the network equations as mentioned above.

3. FRAMEWORK AND ISSUES RAISED

Apart from the theoretical issues, there are several issues concerning the implementation of an active data warehouse. Therefore, in this section, we will start by presenting the general architecture of such a system. In subsection 3.1, we present the grand view for active warehousing and its specific instantiation that we have investigated. Then, in subsection 3.2, we proceed to a detailed presentation of the issues raised within this framework.

3.1. System Architecture

Our architecture consists of the following elements: a Data Source generating data, an intermediate data staging area that will be referred to as the Active Data Staging Area (ADSA) where the processing of data takes place and the Data Warehouse (DW). The architecture is illustrated in Figure 5.

The source comprises a data store (legacy or conventional) and an operational data management system (e.g., a DBMS or an application, respectively). Changes that take place at the source side have to be propagated towards the warehouse, which typically resides in a different host computer. The communication between hosts employs a network protocol (e.g., TCP or UDP). To avoid the extra overhead of overloading the network with half-full packets and, as our experiments indicate, to avoid overloading the source with the extra task of performing this task, we employ a *Source Flow Regulator* (SFlowR) module that compiles changes in blocks and propagates them towards the warehouse.

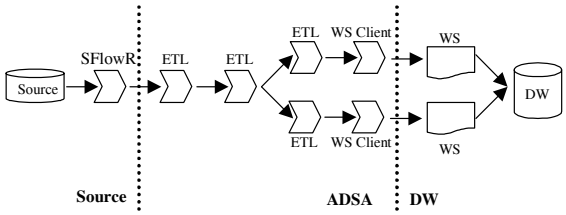

Fig. 5. Architecture Overview

Once record blocks have left the source, an ETL workflow receives them at the intermediate staging area. The role of the ETL workflow is to cleanse and transform the data in the format of the data warehouse. The ETL workflow comprises a set of ETL activities, also called *ETL queues*, each pipelining blocks of tuples to its subsequent activities, once its filtering or transformation processing is completed. In order to perform this task, each ETL activity checks its queue (e.g., in a periodic fashion) to see whether data wait to be processed. Then, it picks a specified number of records, performs the processing and forwards them to the next stage. If less than the specified records exist in the queue, then they are all retrieved. If the queue is empty, then the invocation is postponed, until there exist data to be processed.

The role of the active data staging area is versatile: (a) it performs all the necessary cleansings and transformations, (b) it relieves the source from having to perform these tasks, (c) it can act as a regulator for the data warehouse, too (in case the warehouse cannot handle the online traffic generated by the source) and (d) it can perform various tasks such as checkpointing, summary preparation, and quality of service management.

Once all ETL processing is over, data are ready to be loaded at the warehouse. As already explained, we chose to perform this task through a heavy but reliable (syntactically and operationally) middleware, web services. For each target table or materialized view at the warehouse, we define a receiving web service. To be able to invoke the web service, a client needs to be constructed. To regulate the traffic between the staging area and the warehouse, the client compiles the data in blocks, too. The web service at the warehouse side then populates the target table it serves. Load-balancing mechanisms at the warehouse side and physical warehouse maintenance (e.g., index maintenance) can also be part of this architecture. Still, for the moment, we do not address these problems.

In terms of the particular implementation that we examine in this paper, we have studied the problem as it appears over legacy sources. In our configuration, the source includes two software modules: (a) an ISAM file and (b) an application used to modify data in the legacy data source. In order to manipulate ISAM files, there is a library of ISAM routines that are invoked from the application at the source side. We have modified these library routines in order to replicate the data manipulation commands and send updates towards the staging area. Several ETL queues reside at the staging area performing cleanings, transformations and aggregations. Each ETL activity retrieves data from its queue with a constant rate, retrieving a given number of elements in constant intervals. ETL activities communicate both with each other and with the web service clients via Java thread-safe queues. The transfer from the staging area towards the Data Warehouse is done over HTTP (implying TCP as the underlying network protocol).

For our experiments, we assume that the warehouse simply stores the data performing no other task.

3.2. Issues Raised

In order to fulfill all the goals mentioned in Section 1, using the architectural elements described above, there are some issues raised which mainly concern the tuning and configuration of the system. The key issues that affect system performance and need to be resolved are discussed in this section and classified with respect to their locality at the source or the staging area, as well as the overall setup of the environment. All the technical choices and their alternatives are summarized in Table 1.

3.2.1 Choices concerning the Topology

Having described our architectural elements, the next step is to determine their topology. Our architecture offers the ability of selecting different number of tiers. Several choices exist:

- Two-tier architecture, where the source and the warehouse are found on different machines. There are two alternatives concerning this choice: the first is to place the staging area together with the source, putting the data warehouse on a separate machine. The second alternative is to place the staging area at the host where the data warehouse resides (Figure 6).
- Three-tier architecture, where we use a separate dedicated machine for the staging area, leading to a three-tier topology.

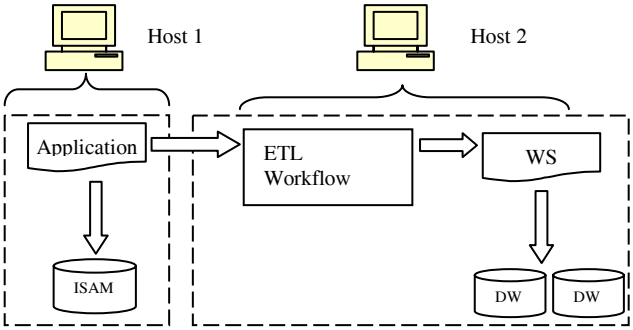

Fig. 6 Two-tier topology: The Data Warehouse and the ADSA reside on the same host, while the Source resides on a separate machine.

Coming to the two-tier architecture, the main issue that arises is related to the placement of the staging area. In the case of the staging area placed at the source, data warehousing operations do not burden the source, but still the resources used by the web services API to perform the invocation remain considerable. A way for dealing with this is to move the staging area to the warehouse host (Figure 6), which can be expected to be more powerful from the source host. This way, the source is completely detached from the active data warehousing process. Naturally, if the warehouse server is too loaded or its configuration too complex for the extra software setup of a web service server, the three-tier architecture can also be employed. Using the three tier architecture solves all the abovementioned problems, but increases the setup and maintenance cost, since an extra server, apart from the one used from the warehouse, has to be engaged and administered.

Having discussed the architectural alternatives for our topology, we can now proceed to discuss the technical issues raised for each of the main components and their overall setup.

3.2.2 Choices concerning the Source

Concerning the source side, the first consideration that arises has to do with the interconnection type between the source and the staging area. Since our goals are to impose as little impact as possible to the source and to make only minor changes, we have chosen the solution of sockets both due to its anticipated (but not thoroughly tested) lighter footprint characteristics and the easiness of programming such a solution.

The next choice is between TCP and UDP protocols for the transmission of data between the source and the staging area. On one hand, TCP offers reliability. On the other hand, UDP offers speed through non-blocking calls, followed by a concern on the server side for the socket buffer size, in case of extended datagram bursts and no reliability.

A third architectural choice concerns the way that changes to the source file are written to the socket, i.e., whether data are organized in blocks before being further propagated to the staging area. There are two ways to deal with this issue: either to write each modification to the socket, or to write bulks of modification commands. In the first case, whenever a data manipulation command is issued, it is immediately written to the socket along with the respective data. In the second case, nothing is written, until a number of records is completed. Then, all records together are sent to the staging area.

3.2.3 Choices concerning the Staging Area

The internal structure of the data staging area and the tuning of its operation are the major issues concerning the performance of our architecture. The staging area is a multithreaded environment with shared components, thus having to be set up properly to avoid race conditions and consistency.

The problem of locking raises the issue of queue emptying rate. Assuming that the input to the staging area is determined by the workload of the source (i.e., it cannot be constrained by the warehouse administrator), a proper emptying rate for the ETL queues has to be determined. A high arrival rate compared to the configured service rate will result in instability and queue length explosion. On the contrary, a very high service rate potentially results in too many locks of the queue (resulting again in delay, contrary to what would normally be expected). It is obvious that the service rate should be close to the arrival rate in order to have both efficient service times, and as less locks as possible.

Another dilemma is related to the interconnection type between the staging area and the data warehouse. As already mentioned, the staging area invokes a web service residing at the warehouse side. Although the SOAP protocol is one-way and asynchronous, implementations abide by the traditional middleware conventions of remote invocation, namely (a) *blocking* and (b) *non-blocking*. Blocking invocation involves an acknowledgment message to be sent from the web service, before its client can continue. In our case, this means that a response from the warehouse is required, delaying however the queue emptying rate. Non-blocking invocation does not delay the queue-emptying process of the web service client, since no response is returned from the invocation.

Finally, the issue of sending data as tuple-at-a-time or blocks is raised again for the communication between the staging area and the warehouse. In this case, apart from the network overhead, the cost of parsing the incoming web service messages at the warehouse plays a role for this choice.

3.2.4 Choices concerning the Warehouse

The data warehouse side is characterized by a web wervice per target table, receiving the cleansed data from the data staging area. The web services API offers three ways of handling the remote invocations of the client that resides in the data staging area. The first way is to create a single web service instance that handles all incoming requests. The second way is to create an instance for every session, and the third is to create an instance for each invocation request. In our configurations, we use the first of these alternatives. The reason is that in our experiments, we have employed one client for the service, which stops its operation after inserting a specific amount of records into the ISAM file. This makes the case of using an instance per session the same as using a single instance. Using an object per request is prohibitive, since we assume high frequency invocations.

Table 1. Architectural choices

Issue	Alternatives
General Architecture	
Topology	- 2-tier, ADSA at the source side - 2-tier, ADSA at the DW side - 3 tier
Source	
Connection Type	- UDP - TCP
Propagation Type	- One at a time - Block-based
Active Data Staging Area	
Interface between the two APIs	- None - Synchronized Queue
Web Service invocation type	- Blocking - Non Blocking
Propagation Type	- One at a time - Block-based
Data Warehouse	
Session management	- Single WS - Instance per session - Instance per request

4. EXPERIMENTS

In this section, we present the experiments we conducted. We present two sets of experiments. The first set presented in section 4.1 deals with the general behavior of the system. The purpose of this set of experiments is to figure out the behavior of each system component separately, and to establish guidelines for building the system. In this case, data are just transferred to the warehouse and no ETL operations are involved. In the second set of experiments, presented in section 4.2, we evaluate the behavior of our system in a realistic setup, based on the conclusions derived from the first set. Naturally, in this case, we also transform data using ETL operations.

Our experimental setup, which stands for both cases, is as follows: The ISAM library that we altered is the PBL/ISAM suite [20] available under GPL license. We have used a sample program distributed within the suite as the legacy application. We use two different data sets for our purposes. The first consists of 100,000 records and the second of 1,000,000 records. The ETL queues of the ADSA have been implemented using the Sun JDK 1.4, whose runtime engine has also been used. As a Web Services platform we have used Apache Axis 1.1 [4] with Xerces XML parser running over Apache Tomcat 1.3.29. Our data warehouse is implemented as a MySQL 4.1 database.

The host we used for the source was a PIII 700MHz with 256MB of physical memory running SuSE Linux 8.1. The host used as the data warehouse was a Pentium 4 2.8GHz with 1GB of physical memory running Mandrake Linux. This server also hosted the staging area. The hosts are interconnected via the switched Fast Ethernet LAN of our department.

Our data were created from the TPC-H data generation tool. For the first case, each row of data has fixed size equal to 20 bytes. In the second case, where we evaluate the system behavior under operational conditions, we used data of variable size. In this case each row has an average size of 140 bytes.

In our experiments we evaluate the cost in marginal conditions. Thus in order to evaluate the worst case, the source stores data at its peak capability. Moreover, since our warehouse host is a much faster computer than the source host, we would not be able to make safe conclusions if we let it operate at full capability (see also subsection 4.1.4). Thus we simulate slower server performance by employing timeouts between operations. This will be explained in more detail later.

4.1. Experiments on Architecture without ETL Processing

This section includes the first set of experiments we conducted. The aim of these experiments is to decide on basic architectural choices of our system. Throughout the experiments, the software operating at the staging area is a simple queue, called *Data Warehouse Flow Regulator* (DWFlowR), receiving source blocks of records and passing them to the warehouse.

4.1.1 Smooth Upgrade

One of the goals of our architecture is to pose minimal modifications to the source's code. In our approach, we do not alter the legacy application itself, but the library that manipulates the ISAM files by adding few lines of code to the routines that are of interest to the purpose of active warehousing. These routines are: the file opening routine, the record insertion routine and the file closing routine. The alterations are located only in the following four points of the library's source code:

1. The first modification is to include our library which contains the socket's client and the SFlowR.
2. The second modification is to add a call to the routine of our library that opens a socket to the staging area at the ISAM file opening routine. This call is performed only if the opening of the ISAM file is successful.
3. The third modification is to extend the insertion routine of the ISAM file library that writes the record to the file with a call to our library's function that propagates the change to the socket. This routine stores the specific record to the

SFlowR's buffer and when the defined number of records is completed, it delivers them to the staging area. Again, this routine is called only after a successful insertion.

4. The fourth modification is to add a call to the routine of our library that closes the opened socket to the staging area, at the ISAM file closing routine. This call is performed only if the closing of the ISAM file is successful.

Figure 7 shows the alterations that we have performed to the library in pseudo-code. The overall length of code that had to be written for this part of the implementation, including the additions at the ISAM library, is approximately 100 lines.

The routine that opens the socket to the DWFlowR reads configuration information from a plain text file, before the opening of the socket. This file contains the following three pieces of information:

1. The number of records the SFlowR will gather
2. The address of the DWFlowR
3. The port of the DWFlowR

Original Routine	Altered Routine

`Open_isam_File()`
`{`
`  …`
`opening_isam_file_commands`
`  …`
`}`
	`Open_isam_File()`
	`{`
	`  …`
	`opening_isam_file_commands`
	`  …`
	`if(open==success)`
	`DWFlowR_socket_open()`
	`}`

`Write_record_to_File()`
`{`
`  …`
`insert_record_commands`
`  …`
`}`
	`Write_record_to_File()`
	`{`
	`  …`
	`insert_record_commands`
	`  …`
	`if(write==success)`
	`write_to_SFlowR()`
	`}`

`Close_isam_File()`
`{`
`  …`
`closing_isam_file_commands`
`  …`
`}`
	`Close_isam_File()`
	`{`
	`  …`
	`closing_isam_file_commands`
	`  …`
	`if(close==success)`
	`DWFlowR_socket_close()`
	`}`

Fig. 7. Code alterations at the routine opening the ISAM file.

As an overall assessment of the impact of our changes, we can say that (a) minimal code had to be written to achieve the replication of incoming updates to the warehouse in an active fashion, (b) simple configuration parameters are required, (c) no changes were required to the code, rather than a simple recompilation under the new library.

4.1.2 UDP vs. TCP

The first parameter that needed to be tested involved the network protocol between the source and the staging area. The goal of our first experiment is to determine the system's behavior using UDP and specifically if there are any datagram losses. The results show a 35% packet loss of data, most probably due to the overflowing of data. Such losses are prohibitive for normal operation of an on-line environment. Therefore, for the rest of the paper, we have fixed TCP as the interconnection protocol between the source and the staging area.

4.1.3 Overhead at the Source

The main requirement for the architecture at the source side involves minimal overhead during regular operation. Therefore, the goal of the next experiment is to measure the overhead that our configuration incurs at the source side. We measure the time to complete the insertion of (a) 100 000 and (b) 1 000 000 to the ISAM file.

First, we measure the effect of using the SFlowR at the source. We try three values: 1, 100, and 1000 records for each packet that the SFlowR sends to the staging area. When using one record at a package, we have in fact the case of not using a SFlowR. In Fig. 8 and 9, we refer to the regular operation of the source (without sending records towards the ADSA) as "plain".

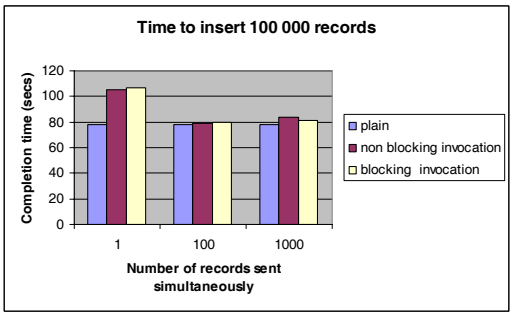

Fig. 8. Time to insert 100 000 records using two-tier topology

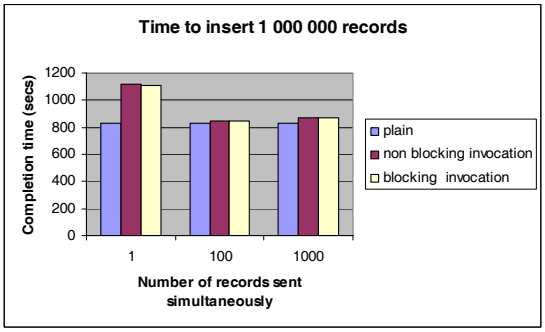

Fig. 9. Time to insert 1 000 000 records using two-tier topology

Another issue worth investigating is the isolation of the Source, ADSA and Data Warehouse layers. Therefore, we employ two modes for the operation of the staging area, to assess its impact. Each test case is examined with blocking and non-blocking invocation for the communication between the staging area and the Web Service at the data warehouse side. The staging area uses a synchronized queue. The input rate at the queue is equal to the output rate of the Legacy Application. The queue's output rate is fixed to one thousand records per second.

Figure 8 depicts the results of the experiment for 100 000 records, while Figure 9 the results for 1 000 000 records. The x-axis for Figures 8 and 9 shows the number of rows in a packet. The y-axis of the diagrams measures the throughput of inserting the records to the ISAM file.

Based on our experimental results, the following observations are made:

1. The SFlowR plays a very important role, since without it the throughput deteriorates by 34%, while using a SFlowR incurs an impact of approximately 1.7%.
2. The way that the DWFlowR is tuned does not affect the source. Regardless of using blocking or non blocking Web Service invocation at the DWFlowR, the source's throughput is the same in both cases.
3. Sending smaller packets of records performs slightly better, since in the case of 1000 records, network propagation time decreases throughput. Moreover, choosing a packet size of 100 instead of 1000 records saves buffer size at the SFlowR.
4. The cost delay ratio in terms of the size of data sent to the warehouse remains stable both in the case of 100 000 and 1 000 000 records.
5. The behavior of our system remains stable regardless of the size of data it has to handle.

4.1.4 Data Freshness

A major requirement in our setting is to achieve the maximum data freshness possible, through our framework. With a 1.7% delay at the source, the focus of interest is isolated in the side of the staging area. The goal of the next set of experiments is to measure the data freshness time provided by our application with respect to the queue emptying rate and the block retrieved from the queue. We consider as *data freshness time* the time required for a record that was inserted in the ISAM file to be transferred to the warehouse.

Specifically, we measure the overall throughput, i.e., the time needed to empty the DWFlowR's queue after the first record is sent to the warehouse. The freshness is then measured as the time needed to empty the queue, which practically stands for the response time for the last record. To perform these measurements, we assume that the legacy application sends 100 000 records to the staging area in blocks of 100 records over TCP. Also, we measure the queue length as an indicator of resource consumption at the staging area.

It is important to determine the behavior of the ADSA using data service rates close to the service rate of the source. Since our data warehouse server is faster than our source, we wanted to simulate slower performance to determine the behavior of the system in marginal conditions. Thus, we empty the queue retrieving the records from the queue using timeouts of 0.1 seconds and retrieving 100, 150 and 200 records each time and then invoking the web service, having as a source data rate approximately 1300 records per second. These are the maximum emptying rates, meaning that if the queue contains fewer records, then all the records from the queue are retrieved. We also present the results of the server operating at its top performance.

The results of emptying the queue using various rates are depicted in Figure 10. In these graphs, two other parameters play a major role. The first parameter, as indicated on the x-axis, is the time required to empty the queue. The second parameter, as shown on the y-axis, is the number of elements in the DWFlowR's queue. Figure 11 depicts the data freshness provided by our architecture.

We measure the time required to transfer all data from the staging area to the data warehouse.

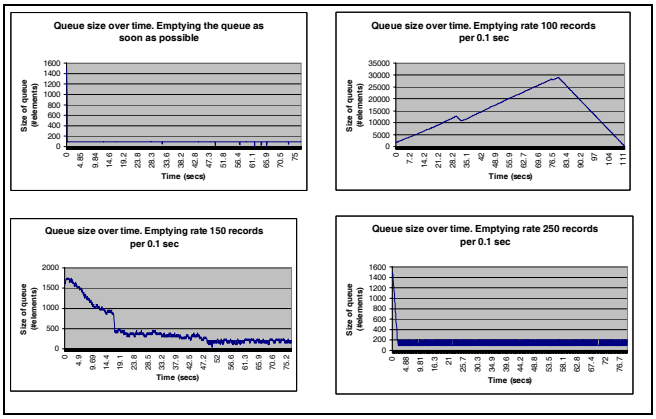

Fig 10 Queue size at the staging area emptying the queue as soon as possible

In Figure 10, the top left graph shows what happens when we let the ADSA operate fully. We can easily see that practically no queue is ever formed. The mean queue size is 100 records which is the rate of the SFlowR. In other words, the ADSA is one step later than the source, in terms of performance.

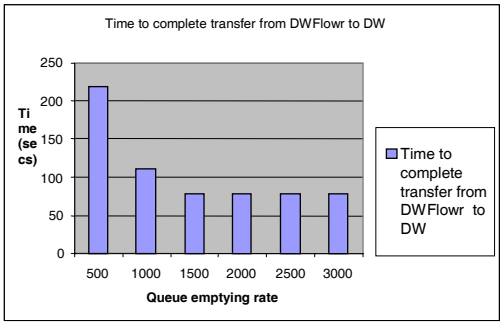

Fig. 11 Queue emptying time at the staging area.

The other three graphs show the queue sizes using service rates of 1000, 1500 and 2000 rows per second. In the first case, where the service rate is lower than the arrival rate, the queue explodes, as expected. In the second case, where we are close to the arrival rate, the queue displays a quite transient yet stable behavior. The last graph practically presents the same behavior as in the first graph even though the service rate is slightly increased compared to the case of 1500 rows per second. We have also experimented with even higher service rates i.e., up to 3000 rows per second, which still present the same behavior. We omit these results due to lack of space.

Observing the results of this set of experiments, we are led to the following conclusions:

1. We can achieve data freshness time equal to data insertion time when we *continuously* empty *a small size* queue.
2. In this case, the size of the queue is equal to the arrival rate from the source, i.e., there is practically no delay at the queue.

4.2. Operational Evaluation

In this subsection, we will use the architectural guidelines derived from the first set of experiments presented in subsection 4.1 to build an active data warehouse where we will also deploy our online ETL operations. The aim of this section is to evaluate the behavior of this fully deployed system.

4.2.1 Impact at the Source

In this paragraph, we will try to refine the results learned in 4.1. For this reason, we examine again the impact on the source system of the packet size of the SflowR. This time we will use small package sizes, as derived from the previous set of experiments.

Figure 12 shows the impact at the source using packets at the SflowR of various sizes. In general, packet sizes of over 25 records offer the least burden to the source. The smallest delay was achieved with a packet size equal to 50, where the source delay was measured to be at 5.8%.

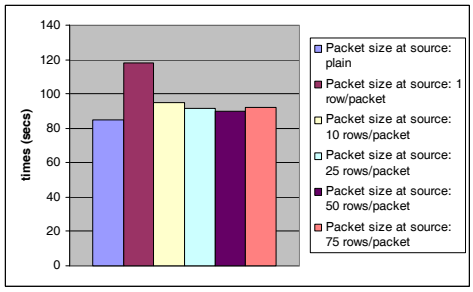

Fig. 12 Packet size of the SFlowR and impact at source

4.2.2 Data Freshness of Online ETL

In this paragraph, we deploy certain ETL scenarios and evaluate their performance compared to the theoretical analysis and in terms of data freshness. For this reason, we consider the following scenarios and their individual steps:

- Scenario (a): We simply transfer data inserted into the legacy application to the warehouse using various service rates.
- Scenario (b): (1) We filter 10% of incoming data through a selection predicate. (2) Then, we employ a surrogate key transformation to the first column of the filtered data. (3) Next, we perform a cumulative aggregation (group by with sum). (4) Finally, data are fed to the warehouse.
- Scenario (c): (1) We filter 10% of incoming data. (2) Then, we additionally filter another 2% of the remaining data. (3) Next, a surrogate key operation is applied to the first column of the data. Then, the stream is replicated along two branches.
    - For the first branch populating a materialized view, (4.1.1) a cumulative aggregation is performed and (4.1.2) data are fed to the warehouse.
    - For the second branch, populating the detailed fact table (4.2), data are fed to the warehouse.
- Scenario (d): (1) We filter 10% of incoming data. (2) We replace the values of the first field, to simulate value computations through functions. (3) A surrogate key transformation is applied. Then, the stream is replicated along two branches:
    - For the first branch, (4.1.1) a cumulative aggregation is performed first and (4.1.2) a filter (HAVING clause) rejecting 6% of the groups is applied. Then, (4.1.3) data are fed to the warehouse.
    - For the second branch, (4.2.1) a second value derivation is performed, (4.2.2) a filter rejecting 2% of detailed input data is applied and, finally, (4.2.3) data are fed to the warehouse.

In Figures 13, 14, 15 and 16 we depict the evolution of the experiments as time passes. The x-axis depicts the time points when we measured the queue length. The final time point gives the time (in seconds) required to complete the transfer from the ADSA to the Warehouse. The y-axis depicts the number of rows existing in the queue. The graphs only show the time points when our measurement showed that the queue is not empty. Each of the queues in the graph is identified by its operation name (e.g., in Figure 13, "FILTER"), possibly its selectivity (e.g., "10" for 10%) and its occurrence in the scenario (e.g., "01" for the first occurrence).

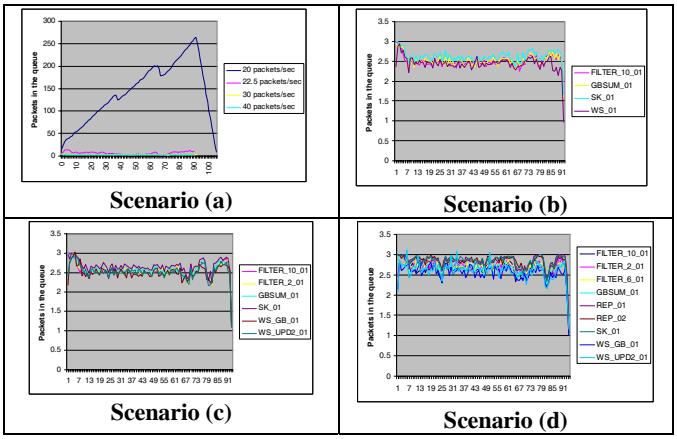

Fig. 13-16 Queues for scenarios (a), (b), (c), (d)

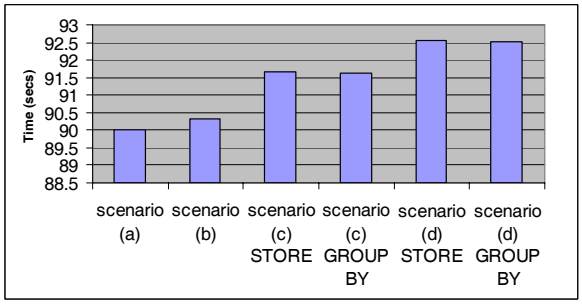

Fig. 17 Data freshness for each scenario

In all the scenarios the block size of the SFlowR was fixed at 50 rows per block. Scenario (a) was configured to use the following service rates: 20, 22.5, 30 and 40 packets per second, which represent rates of 1000, 1250, 1500 and 2000 rows per second respectively. In scenarios (b), (c) and (d) the service rates were simulated to 30 packets per second both for the ETL rates and the Web Service clients.

Finally, Figure17 summarizes the total times needed for the ADSA to transfer all data to the warehouse, for each scenario of ETL queues.

Observing the figures, we derive the following conclusions:

1. The source capability is approximately 1100 rows/sec. In scenario (a) we are led to queue explosion, when we employ service rate smaller than the source's arrival rate. Using a service rate of 1250 rows / sec, which is a setting close to the arrival rate, we can see that transient effects tend to appear, but the queue converges to steady state. By using higher service rates, 1500 and 2000 rows / sec respectively, the queue maintains its steady state.
2. In scenarios (b), (c) and (d) we observe that the entire system, as well as the queue of each operation, maintains a steady state. The number of packets in the queue is less or equal to the maximum number of packets polled simultaneously from the queue. This practically means that after each poll the queue empties and that the ADSA is only one step behind the source.
3. In Figure 17, the total time needed for the entire dataset to be transferred from the ADSA to the Warehouse is dependent on the number of the intermediate ETL operations. As the number of intermediate ETL operations that a packet has to visit increases, the total delay increases as well. Nevertheless, in our exemplary scenarios, the increase is rather small, due to the pipelining of data. The average delay per row is around 0.9 msec for all scenarios.

In Table 2 we present the comparison of our theoretical evaluation of queue length against the observed values. For lack of space, we show only the results of scenario (c) with service rate of 2000 rows/sec; all the other scenarios present identical behavior. As one can observe, in average, the theoretical prediction typically underestimates the average queue length by a very small amount (of the size of 5 records). In our detailed experiments, the system behaves in accordance with this pattern for all four scenarios, with an average error of half a packet (i.e., 25 records).

Table 2. Theoretical prediction vs. actual measurements of average queue length for scenario (c) in packets

	Measured	Theoretical Prediction	Difference
FILTER_10_01	0.160	0.056	0.104
FILTER_02_01	0.134	0.047	0.087
SK_01	0.154	0.054	0.100
GB_SUM_01	0.137	0.048	0.089
WS_GB	0.091	0.031	0.059
WS_GB_UPD	0.100	0.035	0.066

5. RELATED WORK

In this section, we present work related to our approach. Research in ETL has provided results in (a) tools [10, 21], (b) algorithms for specific tasks [7, 15, 16, 18]. Both tools and algorithms operate in a batch, off-line fashion. So far, minimum emphasis has been paid to the investigation of ETL tasks, apart from a general model for [7, 18], where ETL activities are studied under the prism of lineage or resumption of a failed process. As already mentioned, data streams [1, 5, 17] could possibly appear as the paradigm for active warehouse maintenance. So far, streams have been studied from the point of view of continuous querying, without any investigation of transformations or updates. Both our architecture and theoretical analysis could possibly be applied over streams for this purpose. To our knowledge, the only paper related to our approach is [14], where the authors apply a "white-box" (as opposed to our black box) method to determine the properties of SPJ relational operators with respect to queue theory.

Work in materialized views refreshment [12, 13, 24, 25] is orthogonal to our setting. In [13] the authors describe materialized views, their applications, and the problems and techniques for their maintenance. Novel techniques and an up-to-date survey of related work in the field are presented in [12]. Materialized views refreshment fits orthogonally with our on-line refreshment technique, since we can treat each ETL queue as a black-box process. In the context of this paper, a dedicated web service is assigned to each materialized view. Although the tuning of the system for large workloads of views is an interesting topic of research, we find this issue outside the scope of this paper.

Another area related to our approach is the one of active databases. In particular, if conventional systems (rather than legacy ones) are employed, one might argue that the usage of triggers [7] could facilitate the on-line population of the warehouse. Still, related material suggests that triggers are not quite suitable for our purpose, since they can (a) slow down the source system and (b) require changes to the database configuration [6]. In [19] it is also stated that capture mechanisms at the data layer such as triggers have either a prohibitively large performance impact on the operational system. As compared to these problems, our architecture achieves low overhead with minimal impact in the configuration of the source. We conjecture that a replication mechanism similar with the proposed one, propagating log entries towards the warehouse is a possible solution towards this problem.

6. CONCLUSIONS AND FUTURE WORK

Active Data Warehousing refers to a new trend where data warehouses are updated as frequently as possible, due to the high demands of users for fresh data. In this paper, we have proposed a framework for the implementation of active data warehousing, keeping in mind the following goals: (a) minimal changes in the software configuration of the source, (b) minimal overhead for the source due to the "active" nature of data propagation, (c) the possibility of smoothly regulating the overall configuration of the environment in a principled way. In our framework, we have implemented ETL activities over queue networks and employed queue theory for the prediction of the performance and the tuning of the operation of the overall refreshment process. In terms of data freshness, source overhead and minimal impact of software configuration the results seem satisfactory. A summary of the lessons learned is as follows:

- In terms of architecture, isolating the ETL tasks in a special-purpose area, either in the warehouse, or in an intermediate tier, guarantees both minimum performance overhead at the source and the possibility of regulating the flow towards the warehouse target tables.
- Queue theory can be successfully employed as the theoretical background for the estimation of the response of the active staging area. The system reaches a steady state quite close to the predicted behavior. Freshness is quite satisfactory too.

- The overall overhead at the source side is around 1.7% and the amount of code modification is around 100 lines, without affecting applications.
- Tuning the network-related parameters helps. TCP should be used instead of UDP, due to the packet loss of the latter. Organization of rows in blocks, both at the source and the ADSA side increases performance.

Future work includes several directions. A first line of research would have to do with the failure management of the components of the environment, to determine safeguarding techniques and fast resumption algorithms for the event of a failure. Further tuning can be made, by testing multiple concurrent loading sources for the warehouse. Also, the case of materialized aggregate views and schema evolution poses interesting challenges in this context.

7. ACKNOWLEDGMENTS

E. Papapetrou has helped with comments on issues of queue theory and implementation. This research has been partially supported from the European Commission and the Greek Ministry of Education through the Pythagoras Program.

8. REFERENCES

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal, 12(2)*, 120-139, 2003.

[2] G. Alonso, F. Casati, H. Kuno, V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2003.

[3] J. Adzic, V. Fiore. Data Warehouse Population Platform. In *Proc. 5th Intl. Workshop on the Design and Management of Data Warehouses (DMDW'03)*, Berlin, Germany, 2003.

[4] Apache Software Foundation. Axis. Available at http://ws.apache.org/axis/

[5] S. Babu, J. Widom. Continuous Queries over Data Streams. *SIGMOD Record 30(3)*, 109-120, 2001.

[6] Donald Burleson. New Developments In Oracle Data Warehousing. Available at: http://dba-oracle.com/oracle_news/2004_4_22_burleson.htm

[7] Stefano Ceri, Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *Proc. VLDB, Barcelona Spain, September 1991*, 577-589

[7] Yingwei Cui, Jennifer Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal 12(1)*, 41-58, 2003.

[9] W. Duquaine Web Services Ruminations. Presentation at *High Performance Transaction Systems Workshop (HPTS'03)*. Asilomar Conference Center, California, October 12-15, 2003. Available at http://research.sun.com/hpts2003/

[10] Galhardas, H., Florescu, D., Shasha, D., and Simon, E.. Ajax: An Extensible Data Cleaning Tool. *In Proc. ACM SIGMOD*, Dallas, Texas, May 2000, p. 590.

[11] D. Gross, C. Harris. *Fundamentals of Queuing Theory*. Wiley, 3rd Edition, 1998.

[12] H. Gupta and I.S. Mumick. Incremental Maintenance of Aggregate and Outerjoin Expressions. To appear in Information Systems, 2004.

[13] Ashish Gupta, Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *Data Engineering Bulletin 18(2)*, 3-18, 1995.

[14] Qingchun Jiang, Sharma Chakravarthy. Queueing analysis of relational operators for continuous data streams. In *Proc. CIKM*, New Orleans, Louisiana, USA, November 2003, 271-278.

[15] Wilburt Labio, Jun Yang, Yingwei Cui, Hector Garcia-Molina, Jennifer Widom: Performance Issues in Incremental Warehouse Maintenance. In *Proc. VLDB*, Cairo, Egypt, September 2000, 461-472.

[16] Wilburt Labio, Hector Garcia-Molina: Efficient Snapshot Differential Algorithms for Data Warehousing. In *Proc. VLDB*, Mumbai, India, September 1996, 63-74.

[17] D. Lomet, J. Gehrke. Special Issue on Data Stream Processing. *Data Engineering Bulletin*, *26(1)*, 2003.

[18] Wilburt Labio, Janet L. Wiener, Hector Garcia-Molina, Vlad Gorelik. Efficient Resumption of Interrupted Warehouse Loads. In *Proc. of ACM SIGMOD*, Dallas, Texas, USA, May 2000, 46-57.

[19] On-Time Data Warehousing with Oracle10g - Information at the Speed of your Business. An Oracle White Paper. August 2003. Available at http://www.oracle.com/technology/products/bi/pdf/10gr1_twp_bi_ontime_etl.pdf

[20] P. Graf. The Program Base Library. Publicly available through http://mission.base.com/peter/source/

[21] Vijayshankar Raman, Joseph M. Hellerstein: Potter's Wheel. An Interactive Data Cleaning System. In *Proc. VLDB*, Rome, Italy, September 2001, 381-390.

[22] C. White. Intelligent Business Strategies: Real-Time Data Warehousing Heats Up. *DM Peview*, August 2002. Available at http://www.dmreview.com/article_sub.cfm?articleId=5570

[23] A. Willig. Performance Evaluation Techniques. Available at http://www-ks.hpi.uni-potsdam.de/docs/engl/teaching/pet/ss2004/script.pdf, 2004.

[24] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, Jennifer Widom: View Maintenance in a Warehousing Environment. In *Proc. of ACM SIGMOD*, 1995, 316-327.

[25] Xin Zhang, Elke A. Rundensteiner: Integrating the maintenance and synchronization of data warehouses using a cooperative framework. *Information Systems 27(4)*, 219-243, 2002.