Reihe Informatik
001 / 2009

Processes Are Data: A Programming Model for Distributed Applications

Alexander Böhm Carl-Christian Kanne

University of Mannheim
alex|cc@db.informatik.uni-mannheim.de

Processes Are Data: A Programming Model for Distributed Applications

Alexander Böhm
University of Mannheim
Germany
alex@db.informatik.uni-mannheim.de

Carl-Christian Kanne
University of Zurich
Switzerland
kanne@ifi.uzh.ch

ABSTRACT

Applications in distributed environments must scale to an increasing number of concurrently active application instances. Today's application servers spend a significant amount of resources on reliably managing state for these instances, turning them into data management servers instead of process servers.

The goal of the Demaq project is to overcome the limitations of these systems using a novel programming model for applications based on asynchronous messaging (e.g. Web Services). A crucial aspect of our approach is the representation of state. Messages do not only represent requests and replies sent to and from an application, but retained messages are also used to model the application instance state. This contrasts with most of today's application servers where two separate data models, languages and stores are used for requests and state.

In Demaq, a single, highly efficient, reliable message store is used both for requests and instance state, and a single declarative language specifies message flow and state management. This extends data independence to the whole application stack, thereby improving both developer productivity and - as experimental results confirm - application scalability and performance.

1. INTRODUCTION

Applications in distributed environments must scale to an increasing number of concurrently active application instances. Today's application servers spend a significant amount of resources on reliably managing persistent state for these instances, turning them into data management servers instead of process servers. Automatic performance optimization is restricted by a lack of data independence, since the languages used to control the application processes and to specify the individual processing steps are typically not declarative and depend on specific physical representation of the application state (e.g. main-memory Java objects).

.

The Demaq project is an attempt to reconsider the programming model for distributed applications based on XML messaging (e.g. Web Services). Our main goals include the following five objectives.

Efficient State Management without expensive conversion operations that become necessary due to several, incompatible data representations.

High Scalability and Concurrency to support a very high number of concurrently active application instances without state management becoming predominant.

Reliability to support distributed applications that cannot tolerate loss of data, such as business processes.

Optimizability by using a declarative language that allows automatic optimization of execution strategies and storage formats to improve performance.

Programming Convenience by providing developers with efficient means to create XML messaging applications.

To achieve these goals, the Demaq architecture and programming model fundamentally differ from those of today's application servers.

A crucial aspect of the Demaq approach is to model the complete state of running application instances exclusively using messages - there is no other persistent representation of state. A highly efficient, reliable message store is used for data management, and a declarative language specifies message flow and reliability requirements.

Our approach is motivated by the observation that the behavior of a node in a message-driven application is determined by all the messages it has seen so far. The externally visible behavior of the node is represented by messages it sends to other nodes. Hence, the node's processing logic can be specified as a declarative query against the message history, the result being a set of new messages to send. This way, we turn application processing into a declarative query processing problem. In an initial sketch of our vision [10], we focused on the general concepts and language syntax, motivated by simple and elegant application specification and developer productivity. In this paper, we turn towards the performance improvements made possible by our simple message-focused model. We look under the hood of our execution system, reviewing the involved design choices, and introduce techniques that allow to tune the language semantics and execution model to achieve highly concurrent execution and scalability.

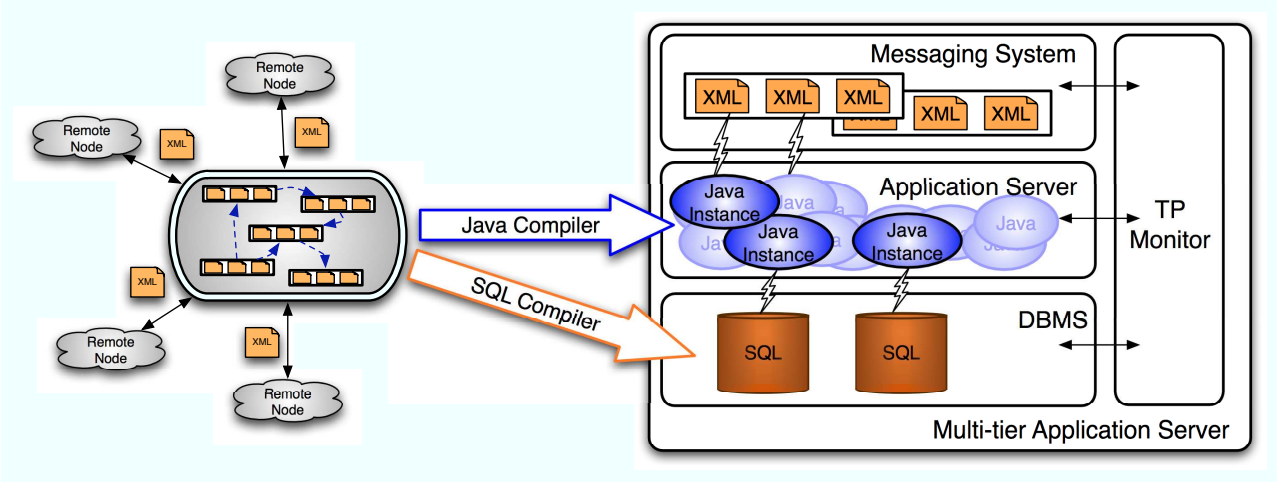

Figure 1: Multiple tiers and data representations in an application server

Our main contributions include

- We describe the syntax and semantics of a rule-based language for declarative XML message processing, based on queries against the message history and a flexible execution model. In particular, we discuss how we intentionally limit visibility of parts of the message history to simplify run-time processing.
- We describe the architecture and implementation of a system that implements our programming model. We explain how the limited visibility and append-only modification of the message history pave the way for highly concurrent transactional message processing without locking,
- We present experimental results comparing our implementation to a commercial application server.

The remainder of the paper is organized as follows. After reviewing related work in Sec. 2, we present the elements of our programming model in Sec. 3. We elaborate on our declarative message processing language in Sec. 4, and discuss the corresponding execution model in Sec. 5. Sec. 6 describes our system implementation. Sec. 7 presents experimental results that show significant improvements in performance and scalability compared to a commercial application server. Sec. 8 concludes.

2. RELATED WORK

2.1 Application Servers

Today, distributed applications are usually executed by multi-tier application servers [2]. For XML messaging applications, these tiers typically consist of queue-based communication facilities (e.g. [19, 22]), a runtime component executing the application logic, and a database management system that provides persistent state storage. An additional transaction processing monitor ensures that transactional semantics are preserved across these tiers. Figure 1 depicts such an architecture.

Application servers allow for the convenient deployment of applications in distributed and heterogeneous environments. However, their use entails several problems which are discussed in literature. Significant functional overlap and redundancy between the different tiers wastes resources [21, 25], and configuration and customization in typical multi-layer, multi-vendor environments with limited native XML support is complex and brittle [2]. Further, frequent representation changes between data formats (XML, format of the runtime component, relational database management system) decrease the overall performance [18].

Apart from simple messaging applications, such as stateless publish/subscribe or message routing, distributed applications require to keep track of their current execution state and related context information. For example, an order application might keep the items in a shopping cart as a customer-specific application state. The runtime components of application servers typically allow for multiple copies - called *sessions* or *instances* - of an application to run in parallel. Each copy materializes its current state information in a corresponding runtime context. Most programming languages (e.g. Java, BPEL or XL [18]) allow instances to access and modify their context using scoped variables.

Management of instances and their corresponding contexts is straightforward in scenarios where only a few instances exist in parallel. However, it quickly becomes problematic if the number of instances increases to the point where the overall size of their contexts exceeds the main memory capacity of the execution system. Consequentially, sophisticated replacement and managing strategies are required, especially for applications which involve long-running activities [8].

2.2 Data Stream Management Systems

Data stream management systems (DSMS) and languages (e.g. [1, 3, 14]) are targeted at analyzing, filtering and aggregating items from a stream of input events, again producing a stream of result items. Several stream management systems rely on declarative programming languages to describe patterns of interest in an event or message stream. In most cases, these languages extend SQL with primitives such as window specification, pattern matching, or stream-to-relation transformation [4].

In contrast to application servers that provide reliable and transactional data processing, stream management systems aim at low latency and high data throughput. To achieve these goals, data processing is mainly performed in main memory (e.g. based on automata [14] or operators [1]). Thus, in case of application failures or system crashes, no state recovery may be performed, and data can be lost. A DSMS may even intentionally lose data, e.g. to remain responsive in periods of high load. In this case, the system may choose to perform load shedding [26] and drop incoming events to reduce the number of messages that need to be processed.

2.3 XML Query and Programming Languages

For an XML message processing system, choosing a native XML query language such as XPath [7] or XQuery [9] as a foundation for a programming language seems to be a natural choice. However, these query languages lack the capability to express application logic that is based on the process state - they are functional query languages with (nearly) no side effects. There are various approaches [11, 13, 15, 18] to evolve XQuery into a general-purpose programming language that can be used without an additional host programming language.

3. PROGRAMMING MODEL

Our programming model describes the application logic of a node in a distributed XML messaging application using two fundamental components.

All application instances operate on the same physical message queues, querying them to recover the application state and updating them by enqueuing new messages. *XML message queues* provide asynchronous communication facilities and allow for reliable and persistent message storage. *Declarative rules* operate on these message queues and are used to implement the application logic. Every rule specifies how to react to a message that arrives at a particular queue by creating another message.

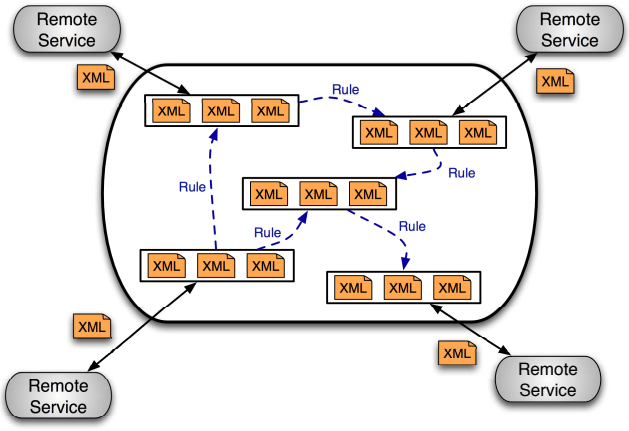

Figure 2: Programming model

An application using this programming model consists of an infrastructure of queues and a set of rules governing the message flow between them (Figure 2).

3.1 XML Message Queues

Distributed messaging applications are based on *asynchronous* data exchange. Queue data structures offer efficient message storage and retrieval operations while preserving the order of incoming data. Queues also allow to decouple the retrieval of a message from its processing. This is particularly useful for an application to keep interacting with communication partners in periods of high load. Additionally, in scenarios involving temporarily unavailable remote endpoints such as mobile devices or sensor nodes, message delivery can be delayed until the remote endpoint becomes available again without blocking the system.

Apart from their typical functionality as intermediate message buffers, our model uses queues as persistent message storage containers that can be queried by application rules. This approach is based on the observation that the state of every application instance in an individual node is derived from the messages sent to and received from its communication partners. Instead of materializing this state in a corresponding runtime context - and maintaining this context for every application instance - it can alternatively be retrieved from the message flow. This approach allows any number of application instances to be supported concurrently without any management overhead, as determining the instance state effectively becomes a query against the message history. As a consequence, messages have to be retained as long as they are necessary to compute the state of an instance.

XML is a very popular message format for distributed applications. This includes applications based on Web Services [2] or Ajax [20], as well as an increasing number of data exchange formats such as RSS or ATOM, and countless domain-specific protocols. Consequentially, data storage in our message queues is based on the XQuery Data Model (XDM) [16]. XDM is the data model of most XML query languages including XPath 2.0, XSLT 2.0 and XQuery. Building on XDM allows us to reuse existing XML processing systems such as stores and query processors without type system mismatches. For our purposes, XDM is particularly suited, as its fundamental type is the ordered sequence, which nicely captures message queue structures.

3.2 Declarative Rules

In our model, the processing logic is specified as a set of declarative rules that operate on messages and queues. Each rule describes how to react to a single kind of event - the insertion of a new message into a queue. Depending on the structure and content of this message, rule execution results in the creation of new messages. These result messages can either become the input for another rule, or be sent to a remote system using queue-based communication facilities.

Our rule language is built on the foundation of XQuery. It allows developers to directly access and interact with XML fragments stored in message queues. Thus, there is no mismatch between the type system of the application programs and the underlying communication format. Additionally, the content of messages and queues can be directly accessed within application programs without crossing system boundaries or requiring complex, intermediate APIs.

In contrast to today's application servers that mainly rely on imperative programming languages, our rule language is *declarative*. Among others, this decision is motivated by the success of declarative, SQL-based languages in stream-

ing solutions [4, 14]. Generally, declarative languages allow for the fast, convenient and efficient development of applications, and often dramatically reduce the development overhead and the required lines of code [23]. They provide execution systems with the freedom to choose from several execution strategies and - compared to imperative languages - with a much greater potential for optimization.

4. LANGUAGE

Every Demaq application consists of four components which we discuss in the following sections. These are message queues, application rules, message properties and user-defined message groups, called *slicings*. To illustrate the practical application of each individual component, we provide several code examples taken from a small online shopping application. We will also use this application in the performance evaluation (Sec. 7).

4.1 Queues

Our programming model incorporates two different kinds of queues. *Gateway queues* provide communication facilities for the interaction with remote systems. There are two different kinds of gateway queues, *incoming* and *outgoing* ones. Messages that are placed into outgoing gateway queues are sent, while incoming gateway queues contain messages that have been received from remote nodes. Queues are also used as persistent storage containers. These *basic queues* allow applications to store messages without sending them to external systems.

As a result, messages received from remote communication endpoints and internal state representation are handled in a uniform manner, thus simplifying application development.

Example: Basic Queues.

The following queue definition statements create three *basic* queues. These are used for persistent, local message storage of customer master data and the items ordered by customers (books and music).

```
create queue customerMasterData kind basic mode persistent;
create queue bookCart kind basic mode persistent;
create queue musicCart kind basic mode persistent;
```

Example: Gateway Queue Definition.

The example below creates a gateway queue that receives messages from external HTTP clients on port 2342. An application can reply to such a request via the `outgoingMessages` queue. Any messages placed in the outgoing queue is automatically correlated to an input message, and sent to the initial requester as a synchronous response. For asynchronous protocols, no response queue is needed.

```
create queue incomingMessages kind incoming
  interface "http" port "2342" response outgoingMessages
  mode persistent;
```

4.2 Message Properties

Every message in our system is an XML fragment that was either received from an external source or generated by local application rules. Apart from their XML payload, messages are associated with additional metadata annotations that are kept separate from the XML payload. These *properties* are key/value pairs, with unique names as their key and a typed, atomic value. They are determined during creation and remain fixed over the entire lifetime of a message.

There are several ways how a property can be associated with a message:

Explicit A property value may be explicitly set by an application rule when enqueuing a message. Explicit properties allow developers to annotate messages with additional information without modifying their XML body.

System Several properties are set by the system, such as creation timestamps or transport protocol information (e.g. the original message sender).

Computed A property value may be computed from the XML payload of a message. These properties are comparable to views in database systems, which provide aliases for frequently used expressions.

Example: Property Definition.

The following two properties are used to *compute* the customerID or transactionID from the content of the messages stored in the queues created above. This is done by evaluating the corresponding path expression. The `fixed` modifier indicates that these properties may not be set explicitly.

```
create property customerID
  queue customerMasterData fixed value //customer/ID/text();
create property transactionID
  queue bookCart, musicCart fixed value //transactionID/text();
```

4.3 Slicing the Message History

Conceptually, the state of application instances in Demaq is encoded in the message history, and rules access the state by posing queries against the history. Of course, processing queries against all existing messages for every processing step is inefficient, and the need to filter the relevant messages for every rule may lead to repetitive code in rule bodies. In addition, keeping the complete message history forever requires unbounded storage capacity.

For these reasons, the Demaq language provides mechanisms to declare portions of the message history that are relevant in particular contexts, called *slicings*.

They can be seen as a kind of parameterized view [27] that extends the concept of data independence to the application state. Slicings support compact rule formulation by giving a name to frequent parameterized expressions. Further, the explicit declaration of relevant message subsets can be used for optimization purposes, e.g. by indexing or materializing slicings.

Slicings are used to simplify the implementation of recurring design patterns in messaging and workflow applications, such as "correlation sets" in BPEL , or "conversations" in XL [18]. Additionally, they can be used to join control flow after executing several tasks in parallel, or to establish synchronization points and milestones [28] within an application.

4.3.1 Slicing Definition

Figure 3 illustrates different access patterns in our example application. In the depicted example, two different queues are used to store incoming orders by their message type. Additionally, the application needs to access messages on a per-customer basis, independent of their queue-based, physical storage location.

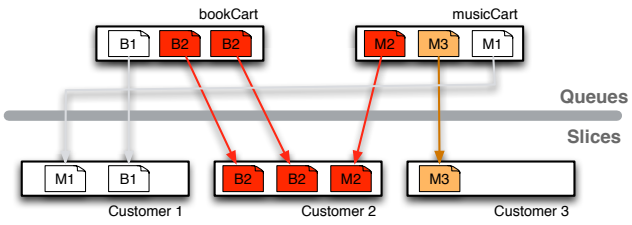

Figure 3: Groups of logically related messages

A slicing defines a family of *slices*, where each slice consists of all the messages with the same value for a particular part of the message (*slice key*). In the example (Figure 3) above, this slice key is a unique, customer-specific identifier. To identify the part of a message that should be used as the slice key - and thus as the basis for the partitioning - we rely on the property mechanism introduced in Sec. 4.2.

The use of property values as slice keys avoids additional language primitives. It reflects the fact that the way property values are defined nicely matches the criteria according to which applications need to group messages in slices. Just as in the property example above, slice keys sometimes need to be computed from the message. In other cases, the rule creating a message might want to specify the target slice by explicitly setting the slice key.

A slicing is created by specifying a unique name and the *slicing property*. The property definition lists a number of queues on which the property is defined. Messages from these queues are partitioned into *slices* according to their property value. All messages that share the same value of the slicing property become part of the same slice.

Example: Slicing Definition.

In this example, two different slicings are created. The `masterDataForCustomer` slicing allows to retrieve all messages from the customerMasterdataQueue that belong to a particular customer.

```
create slicing masterDataForCustomer on customerID;
```

By using the `cartItemsForCustomer` slicing, all order messages for an individual customer transaction can be retrieved.

```
create slicing cartItemsForCustomer on transactionID;
```

4.3.2 Relevant Slice Suffix

Slicings declare partitions of the message history which are relevant to the application. A simple, value-based partitioning is not enough, however, as it would still require to retain the complete, unbounded message history. Instead, we need some kind of mechanism to specify which messages reflect the relevant application state and need to be accessible to rules, and which messages have become irrelevant and may thus be dropped to save space.

To avoid unbounded buffering of message streams, *windows* have been proposed to specify relevant sub-streams [1] based on their position in a stream. The boundaries of such windows are based on the window size or relative to some landmark object in the stream, and the application developer must translate the message retention needs into window specifications.

In Demaq, we allow application developers to directly specify a condition that must be met by the messages that are sufficient to represent the current application state. Access to the slice then yields the smallest suffix which contains such a set of relevant messages. This very powerful semantics captures existing window types (see below) and at the same time allows for a very intuitive, direct expression of relevance conditions from the application domain.

For example, an application rule that performs order processing is only interested in those messages belonging to a customer order that has not been completed yet. To filter out unnecessary messages, a slice can be declaratively constrained using a `require` expression.

The require expression is an arbitrary XQuery expression of type boolean. Among all the contiguous sets of candidate messages in the slice that fulfill this condition, the most recent set is considered the currently relevant state of the slice. This set, and and any messages more recent than that, are visible to the application (e.g. when using the `qs:slice` function explained later). The require expression may refer to the candidate set of relevant messages using a special function (`qs:retainedMsgs()`, see below). However, it may not refer to any other messages in the system. The reason for this latter constraint is simple, efficient evaluation and garbage collection.

Since the require expression may not refer to other parts of the system state, and it always includes a complete suffix of the message history, and the fact that we chose the most recent qualifying set, guarantees a monotonous behavior of our relevant slice state: We can divide the slice into two parts, a relevant suffix (marked gray in Figure 4) and an irrelevant prefix. Our semantics guarantees that, once a message belongs to the irrelevant prefix of a slice, it will never become relevant again. In other words, the boundary between relevant and irrelevant messages only moves toward more recent messages. Thus, it can be represented and tested using a simple message identifier or timestamp comparisons. This allows a simple, decoupled garbage collection strategy: If a message is no longer visible to any application rule because it is not part of any relevant suffix, it can be safely pruned from the message history. Consequentially, storage capacity can be reclaimed in a separate garbage collection process that never conflicts with rule evaluation.

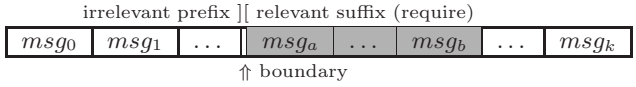

Figure 4: Message history in a slice of size k+1

Example: Require Expression.

In this example, only the last five messages in the slice (accessed using the `qs:retainedMsgs` function) are relevant (this corresponds to a sliding window).

```
create slicing lastFiveCustomerOrders on bookCart
  require count(qs:retainedMsgs()) gt 5;
```

In the following example, the require expression is used to filter out all completed customer transactions. Only transactions that have been started and have not terminated yet (indicated by a message containing a `stop` element) will be returned.

```
create slicing unfinishedTA on transactionID
  require qs:retainedMsgs()//start
```

```
and not(qs:retainedMsgs()//stop);
```

If no suffix matching the require condition can be found, the entire message history is returned. Thus, as in the example below, a require condition of false() can be used to provide an application rule with access to the complete message history. However, this condition has to be used with care as it requires the entire message history of the slice to be preserved. Thus, the system might eventually run out of storage capacity.

```
create slicing completeHistory on bookCart require false();
```

4.4 Application Rules

For our application rules, there is a significant overlap with the capabilities of existing, declarative XML query languages, in particular with XQuery [9]. XQuery allows for querying (sequences of) XML documents, document construction, supports XML data types, schema validation, etc.

Building on an existing language, such as XQuery, provides significant advantages. Developers can benefit from previous experience and reuse programming tools and development infrastructure. Additionally, existing query processing techniques can be potentially adapted to our application language. In the following sections, we discuss how the features required for building message-driven applications can be integrated into XQuery.

4.4.1 Assigning Application Rules

Every application rule is assigned to a single queue or slicing. Whenever a message gets inserted into this queue or slicing, the rule is evaluated with this message as the context item. In order to allow application developers to perform this association of rules to queues and slicings, we extend XQuery by incorporating an additional rule definition expression. Rule definition expressions can be used to give a unique name to an XQuery expression and assign it to a particular queue or slicing.

Optionally, an *error queue* can be defined for an application rule. Whenever a runtime error is encountered during the execution of this rule, a corresponding notification message is sent to this queue. Thus, this error handling mechanism allows other rules to handle this error. If no error queue is defined (as in the examples below), error notifications are inserted into a system-provided, default error queue.

Example: Application Rules.

The rule definition statement below specifies how to react to messages that arrive at the incomingMessages gateway queue from external sources. Whenever a message is inserted into this queue, the rule (named `registerNewCustomer`) checks whether the message contains a particular XML element (`registerNewCustomer` in line 2). If this element is found, the rule forwards the message to the `customerMasterData` queue (line 4) and sends back a confirmation to the caller (line 5). The `enqueue message` statement is used to enqueue the messages into the two corresponding queues.

If the message triggering rule execution does not contain a `registerNewCustomer` element, the rule does not perform any action, as the empty result of the `else` branch in line 8 indicates.

```
1 create rule registerNewCustomer for incomingMessages
2 if(//registerNewCustomer)
3 then(
4   enqueue message . into customerMasterData,
5   enqueue message
6     <result>Inserted customer masterdata</result>
7     into outgoingMessages )
8 else ();
```

4.4.2 Enqueuing Messages

Every application rule describes how to react to a message by creating new messages and enqueuing them into local or gateway queues. While XQuery allows for the creation of arbitrary XML fragments, it does not incorporate any primitives for performing side effects. In our model, this is a severe restriction, as there is no possibility to modify the content of the queues underlying our application rules.

The XQuery Update Facility [12] aims at eliminating this limitation by allowing for the declarative specification of updates on instances of the XQuery data model. For this purpose, the Update Facility extends XQuery with new primitives that represent pending update operations. Expressions which return such pending updates are called *updating expressions* and can be combined by using existing XQuery constructs such as FLWOR or path expressions. Updating expressions produce a list of pending update primitives that are applied after the entire expression has been evaluated, thus resulting in a snapshot semantics for expression evaluation.

We adopt the extensions proposed by the XQuery Update Facility to perform side effects on the messages store. Every application rule is an updating expression that produces a (possibly empty) list of messages that have to be incorporated into the message store by enqueuing them to corresponding queues. In order to allow application programs to both specify the XML fragment to be enqueued as well as their target queue, we extend the XQuery Update Facility with an additional `enqueue message` update primitive.

4.4.3 Message Access

While XQuery incorporates powerful features to query (sequences of) XML documents, it does not provide operations for accessing the content of structures such as queues, properties and slices. These read-only access operations can be easily provided by the runtime system in the form of external functions, particularly without requiring changes to the syntax or semantics of XQuery. In application rules, they are used to access the sequence of XML messages stored in a queue or slice using their unique identifier as a key. Our language incorporates the following functions:

- *qs:queue* can be used to retrieve all messages stored in a particular queue of the system. It takes a single parameter referencing the name of the queue that should be accessed (e.g. `qs:queue("bookCart")` can be used to access all messages stored in the bookCart queue).
- The *qs:slice* function allows to retrieve all messages belonging to a slice. Its two parameters are the name of the slicing and the slice key (e.g. `qs:slice(2342, "cartItemsForCustomer")` can be used to access all messages in the cartItemsForCustomer slicing with a slice key of 2342).
- The *qs:slicekey* function can be used to retrieve the slice key for a message with respect to a given slicing. It's two parameters are the slicing name and a message

(e.g. `qs:slicekey("cartItemsForCustomer",.)` can be used to access the slice key of the context item with respect to the cartItemsForCustomer slicing).

- *qs:property* allows developers to access the value of a property with a given name of a particular message (e.g. `qs:property("customerID",.)` can be used to retrieve the content of the customerID property of the context item).
- The parameterless *qs:message* function can be used to access the message triggering the execution of a rule.

Example: Slicing Access within Rules.

The example below illustrates how system-provided access functions can be used in application rules. Here, whenever a checkout message is received for a particular customer transaction, the `qs:slice` function is used to retrieve the last version of the master data for a particular customer (line 7), as well as all items ordered in the context of the current customer transaction (line 9).

This data is used to compile a delivery confirmation containing all ordered items as well as the last known delivery address for this customer, and to send the confirmation to the customer using a gateway queue.

```
1  create rule handleCheckout for incomingMessages
2  let $request := //checkout
3  return
4    if($request) then
5      let $transactionID := $request/transactionID/text()
6      let $customerID := $request/customerID/text()
7      let $customerMasterData := qs:slice($customerID,
8          "masterDataForCustomer")[position()=last()]
9      let $customerOrders :=
10       qs:slice($transactionID, "cartItemsForCustomer")
11     let $result :=
12       <result>
13         <orderedItems>{$customerOrders//item}</orderedItems>
14         <delivery>{$customerMasterData//address}</delivery>
15       </result>
16     return enqueue message $result into outgoingMessages
17   else();
```

5. EXECUTION MODEL

The Demaq language provides simple, yet expressive primitives to describe desired reactions to messages in terms of the message history. The use of a declarative language for rule bodies allows data independence and efficient execution using a query optimizer.

Our objective is to create an elegant way to completely specify stateful messaging applications, and not only to monitor or analyze message streams - we not only want to read state, but to modify it. Hence, a crucial aspect of the Demaq design is to define how state can be managed in a reliable way and - at the same time - allow for an efficient and scalable application execution.

The design issues in this context revolve around the transactional coupling of rule execution to the message store. It turns out that modeling both requests and state information as messages yields novel opportunities to improve execution performance. A major reason for this is the append-only strategy for the message history: We never perform in-place updates. As a consequence, there is much less need to synchronize concurrent execution threads, and there are fewer ways how a concurrent modification of the system state can cause conflicts.

The Demaq execution model captures the typical behavior of message-driven applications in a few simple rules and guarantees which are observed by the Demaq run-time system. Our model is natural because it mirrors the typical architecture of existing messaging applications - a simple processing loop. It precisely determines Demaq rule semantics, and at the same time leaves enough freedom for an actual implementation to optimize run-time performance, as we will see in Sec. 6.

5.1 Core Processing Loop

The fundamental behavior of messaging applications can be described as a simple loop that (1) decides which message(s) to process, (2) determines the reaction to that message based on the message contents and application state, and (3) effects the reaction by creating new messages. In existing systems, this loop is mostly coded by hand, optimizing for the requirements of each application. Actual implementations of the Demaq model may use any form of processing loop(s) that obeys the following constraints:

1. Each message is processed exactly once. This means that the evaluation of all rules defined for the message's queue and slicings are triggered once for every message.
2. Rules are evaluated by determining the result of the rule body as defined by XQuery (update) semantics, extended by the built-in function definitions described in Sec. 4.4.3. The result is a sequence of pending update operations in the form of messages to enqueue.
3. The overall result of rule evaluation for a message is the concatenation of the pending actions of the individual rules in some non-deterministic order.
4. Processing the pending actions for a message is atomic, i.e. after a successful rule evaluation all result messages are added to the message history in one atomic transaction, which also marks the trigger message as processed.
5. All rule evaluations for the same trigger message see the same snapshot of the message history, which contains all messages enqueued prior to the trigger message, but none of the messages enqueued later.

This list includes strong transactional guarantees necessary to implement reliable state-dependent applications, but still allows many alternative strategies to couple message processing to a transactional message store. We discuss the implications below.

5.2 Transactional Coupling

The main model discussed above explains the desired transactional properties our application engine must guarantee when accessing and modifying the message history. To better understand the design decisions for our engine, we review some of them in terms of the classification by Paton and Diaz [24] for active databases:

single message transition granularity Every message is subject to a separate processing iteration, which simplifies rule semantics. The focus of XQuery rule body evaluation is always a single message, matching the XQuery concept of a single context item.

detached event-condition coupling mode and **iterative cycle policy** The messages may be processed in a separate transaction from their creation. This is demanded by our asynchronous environment and allows many unprocessed messages to reside in the system at any time, increasing parallelization opportunities and scheduling flexibility.

deferred condition-action coupling mode As long as every message is eventually processed, we allow some delay between processing the message and adding the result messages. Again, this improves concurrency and scalability.

all parallel rule scheduling strategy All rules for a single message see the same state of the message history. This is motivated by the interpretation of each rule as an isolated statement of fact about the system behavior – if a certain situation arises, a certain action will eventually happen, no matter what other rules are defined for the same situation. This strategy also allows to factorize common subexpressions across several rules.

Note that the above model does not allow for message store transactions that span rules. However, application developers do have some control over the amount of decoupled, asynchronous execution: The expressive power of XQuery allows the bundling of complex processing steps into single rules, which are executed in a single transaction and hence allow to constrain the visibility of intermediate results to concurrent transactions. Further, application developers can isolate intermediate messages in local queues that are not accessed by conflicting control paths.

Another implication of our execution model is highly simplified concurrency control, which we discuss in a separate section below.

5.3 Enqueue-Time Snapshot Isolation

To achieve a maximum of concurrency, our message store uses a variant of snapshot isolation [6] that is made possible by our unified message-based view of state and requests. In general, snapshot isolation freezes the system state visible to a transaction by creating a private version of the state. This avoids locking and improves concurrency, but requires the retention of old state versions and conflict resolution policies.

In our programming model, guaranteeing snapshot isolation is very cheap, because management of old state versions is trivial: there are no in-place updates - we can access old versions of the system state just by ignoring newer messages. We simplify this by using as begin-of-transaction (snapshot) time for our rule evaluation the enqueue-time of the trigger message. Hence, rule evaluation can only see all messages enqueued prior to the trigger message. Concurrent rule evaluation transactions do not need to lock parts of the history, because updates by definition do not affect their visible part of the message history. We only need to synchronize the message writing transactions to guarantee atomic insertion of result messages. Note that deadlock handling is simple, as the complete set of updates is known before the first update needs to be performed. This strategy tremendously improves the concurrency and scalability of our application engine, because very few short-term locks are required for synchronization.

5.4 Message Scheduling

Message scheduling in Demaq decides on the order in which messages are processed. This processing order is only deterministic for messages in the same slice, which are processed in their enqueued order. The rationale for this decisions is that slices are logical primitives close to the application domain (e.g. messages for a single customers) where out-of-order processing would confuse developers. In all other cases, we allow the run-time system to schedule processing in any order. This can be used to improve locality of access, or to control quality of service.

5.5 Error Handling

There are many sources of errors that may prohibit a successful application execution. Examples include, but are not limited to dynamic XQuery errors, communication failures, and resource exhaustion. In case of errors, rule evaluation is aborted, and none of the resulting messages is inserted. Further, every error is reflected by a corresponding error message conforming to a standardized XML error schema, which is enqueued to designated error queues specified in the program. This allows developers to react to error conditions using rules that provide a contingency plan.

6. SYSTEM

In this section, we outline the architecture and implementation of the Demaq system that implements our programming model (Figure 5).

When deploying a Demaq application, the rule compiler is used to transform the application specification into execution plans for the runtime system. The runtime system consists of three major components. A transactional XML queue store provides efficient and reliable message storage. Remote messaging and transport protocol aspects are handled by the communication system. The rule execution engine executes the plans generated by the rule compiler.

6.1 Rule Execution Engine

The rule execution engine implements our execution model described in Sec. 5. At application startup, this includes setting up the message store, the initialization of the communication system, and the creation of main-memory data structures and processing threads. At runtime, governed by the execution model defined in Sec. 5, it decides when and in which order messages from the queue store should be processed, how to incorporate updates, and when to send or receive messages from the communication system.

System Startup.

For new applications, physical message queues and corresponding index structures are created. A rule compiler optimizes application rules and transforms them into query execution plans which are also written to the message store.

Starting from such a persistent representation of an application, the actual startup of a node is as follows. First, the transactional message store is started, performing any necessary recovery on the queue and index structures. The rule execution engine then reconstructs the contents of its main-memory schedulers from the persistent state of the message store by querying for unprocessed messages. Next, the communications subsystem is initialized, and the gate-

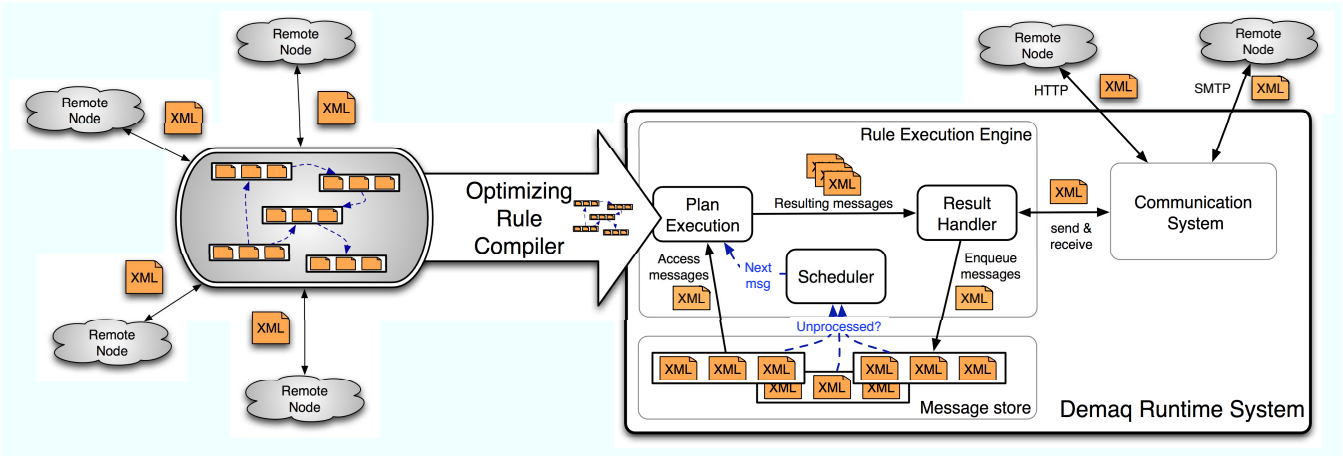

Figure 5: Demaq system architecture

way queues are connected to the network. Finally, message processing is started.

Rule Body Plans.

For each queue, the execution engine keeps a single execution plan for the rule bodies of all rules that must be evaluated for a new message in that queue and the associated slicings. These plans are partly executed within the execution engine, and partly pushed towards the query execution engine of the underlying storage engine. The plans also merge the pending results of all rules into a single pending action list.

Concurrent Message Processing.

The rule execution engine maintains a pool of concurrent processing threads, which repeatedly perform the following processing cycle (visualized in Figure 5).

Each iteration of the cycle consists of selecting the next message to process, rule execution and, finally, the incorporation of the results into the corresponding queues. Currently, all these operations are performed in the context of one atomic transaction against the message store. As the message store uses snapshot isolation for concurrency control, several processing threads can simultaneously operate on the same queues and slicings without blocking and interfering with each other. Processing a message consists of running the execution plan associated with the queue (see above). This evaluation results in a list of pending actions, mostly consisting of new messages to create. These unprocessed messages are stored in their target queues, ending the cycle.

Slicing Management.

Our implementation uses secondary B-Tree index structures to materialize slices. For each slicing, a seperate index is created. Using this index, the runtime system can efficiently access the messages for a slice key without expensive queue scans.

In addition to an index access based on the slice key, the query sub-plans for slice access include filtering operators according to the require expression. A significant performance improvement can be achieved by remembering the current boundary of the relevant suffix for each slice key – due to the monotony explained in Sec. 4.3.2 we only need to examine candidate sets that are more recent than the current boundary.

Message Garbage Collection.

To recover storage capacity, the runtime system includes a message garbage collector that may operate as a background process. The garbage collector checks for each message whether it is required by at least one slicing according to it's `require` expression. If the message is no longer required by any slice and has been processed, it can be safely deleted from the message store without altering the runtime behavior of the system. In this way, the garbage collector regularly prunes the message history and reclaims storage capacity without interfering with concurrent rule execution.

6.2 Rule Compiler

The purpose of the rule compiler is to transform applications into execution plans for the runtime system. After verifying the syntactical and semantical correctness of an application, it is normalized. Normalization e.g. involves removing syntactic sugar such as rules defined on slicings (which can be substituted with equivalent rules defined on queues). In a next step, the compiler tries to optimize the application by applying rewriting heuristics.

Optimization opportunities exist on several levels of an application.

Rule-set rewriting We can change the overall structure of an application by modifying its set of rules. For example, the compiler merges all rules defined on the same queue into a single, combined rule. This simplifies factorizing common subexpressions across rules and saves the runtime system from invoking the rule execution component multiple times for a single message.

Rule-body rewriting We can rewrite the body of individual rules, e.g. by inlining access to fixed, computed properties, or merging computation of the require expression with rule-body evaluation.

XQuery optimization A significant part of our programming language consists of the XQuery Update Facil-

ity, and many optimization techniques developed for XQuery can also be applied to our application rules. To profit from these techniques without reimplementing all of them, the compiler can split rule bodies into two parts, one processed by the Demaq rule execution engine, and one processed by the message store. In case of XQuery-enabled message stores (as is the case in our current implementation), the store-processed part is simply rewritten into an XQuery expression without Demaq-specific constructs, which can then be optimized by the store's XQuery compiler.

Physical optimization Other platform-specific rewrites may be performed to speed up application processing. For example, in our runtime system, there is a special operation which works similar to a link in Unix file systems. This operation avoids a full copy of the message when messages are forwarded unchanged from one queue to another. The query compiler can replace `enqueue message` with this operation (called `enqueue link`) where possible.

Finally, after the normalization and optimization rewrites have been applied, the application is transformed into an intermediate XML representation. It includes the remaining queue, property and slicing definitions that were not optimized away by the rule compiler, as well as execution plans representing the application logic for the remaining rules. This intermediate XML encoding is used to initialize the individual components of the runtime system that will be discussed in the next sections.

6.3 Transactional XML Queue Store

Our message store is built on the foundation of native XML base management system. Our current implementation alternatively uses Natix [17], a research prototype of a native XML data store, or IBM DB/2 Version 9.

Natix organizes XML document repositories as collections, which are unordered sets of XML documents. As our programming model requires queue-based message storage, we had to extend the existing, collection-based data handling and recovery facilities to support a queue-based management model. These extensions allow us to use the efficient XML storage facilities as well as the sophisticated recovery and schema management features of Natix for our message queues. Most importantly, we use the XQuery interface of the system to efficiently evaluate rule bodies on the message queues. Natix supports creating persistent B-Tree indexes, which we use to implement slicings (see Sec. 6.1), and our specialized version of Snapshot Isolation (see Sec. 5.3).

As an alternative to Natix, our runtime system may optionally use IBM DB/2 as the underlying database management system. In this case, as DB/2 does not incorporate native queue support, auxiliary tables are used to simulate queue-based storage (we are currently investigating how to best represent our queue semantics in a relational system).

6.4 Communication System

The communication system provides all remote communication facilities. It implements both asynchronous and synchronous transfer protocols (such as HTTP and SMTP), and thus allows applications to interact with various types of external communication endpoints.

The design goal of the communication system is to simplify messaging operations for application programs. In the best case, remote communication becomes as simple as enqueuing a message into a local queue. This becomes possible as the communication system hides protocol-specific operations from developers and the other components of the runtime system wherever possible.

6.5 Visual Editor

To further increase developer productivity and allow for the convenient specification of Demaq applications, we have created a visual editor (Figure 6). The editor allows to quickly set up the queues of an application using a simple drag-and-drop mechanism. Rules governing the message flow can be "drawn" between the queues, while the rule body is written using a syntax-aware editor with online syntax validation. Additionally, users may easily define and assign message schemas using an integrated XML Schema editor.

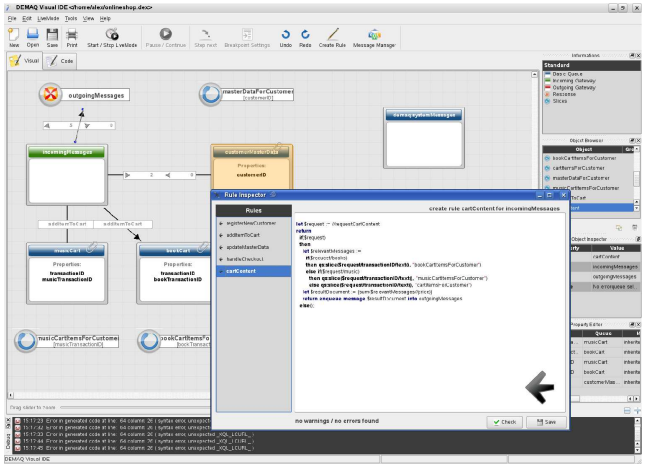

Figure 6: Visual Editor

7. EXPERIMENTS

In this chapter, we provide a brief experimental evaluation of our programming model. For this purpose, we use the Demaq runtime system introduced in the last section to execute an exemplary online shopping application. Several parts of this application are shown in the examples of Sec. 4.[1]

We also implemented an application with equivalent functionality as a BPEL process and executed it on a commercial, enterprise-class application server with a relational database back-end. Unfortunately, licensing restrictions do not allow to disclose additional details about the system, let alone vendor name and software version. However, we believe that the results help to evaluate the performance of our implementation in the light of one of the most advanced application servers available today.

[1]The complete Demaq application specification, the templates used for generating test requests as well as the equivalent BPEL application code is available at http://www.demaq.net/.

7.1 Setup

All measurements were performed on a server equipped with an AMD Athlon 64 X2 Processor 4600, 2 GB of main memory, running Opensuse Linux 10.3. This system was used to run the BPEL application server and our native runtime system. An additional client computer was used to send messages via HTTP.

The Demaq runtime system consists of about 30,000 lines of C++ code. For reliable and persistent message handling, it integrates Natix version 2.2 with our queue extensions. The rule execution component is based on an open-source XQuery processor.

In order to get an impression of the runtime to expect during the following measurements, we first performed an exemplary run of our online shopping application, consisting of 24 messaging operations: The client connects to the server, adds both 10 books and music items (each reflected by a message of 2.5 KB) to the shopping carts, requests the total value of both music and book items, and finally performs a checkout operation. This run was repeated 100 times to reduce the effects of statistical outliers.

The application server required an average of 6.25 seconds in order to run the scenario. Using our native implementation, the same run took 2.18 seconds, which confirms that native XML data handling and avoiding a multi-tiered architecture can help to improve application runtime.

7.2 Performance Impact of Context Size

In order to investigate the impact of context size on the runtime performance, we subsequently add 10000 books (each 2.5 KB in size) to the shopping cart of a single application instance. While a single customer buying thousands of books is rather unlikely for our online shopping example, handling thousands of messages in a single application context is no uncommon scenario in other application domains [5].

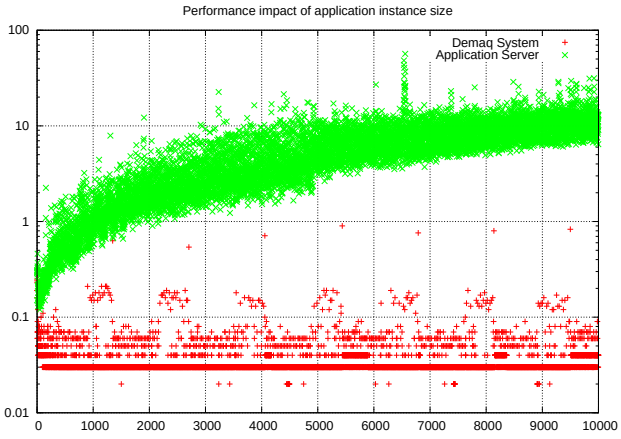

Figure 7: Performance impact of context size

Figure 7 visualizes the round-trip-time (in seconds) for each request adding another book to the shopping cart. With growing instance size, the response times of the application server deteriorate. This effect might be caused by performing an in-place update on the corresponding data structure of the runtime context and writing it back to the database back-end. In our runtime system, every additional book can be appended to a queue of the system, thus leaving the response time virtually unaffected and below one second.

7.3 Parallel Application Instances

In this experiment, we investigate the impact of multiple concurrent, active instances, each of them storing 100 book orders (250 KB) and 10 music orders (25 KB). We analyze how the response time of the systems change with an increasing number of parallel instances. For this purpose, our client sequentially requests the server to calculate the overall price of the music order items for each instance. In order to reduce the effects of statistical outliers, all measurements were repeated 100 times.

Figure 8 depicts the average response time (in seconds) for answering a client request. An increasing number of active instances has a considerable impact on the response times of the application server performing expensive instance management operations. For our runtime system, there are no instances that need to be managed. Instead, all messages that belong to a particular context are retrieved by querying the message store. Thus, an increasing number of parallel instances does not interfere with the response time.

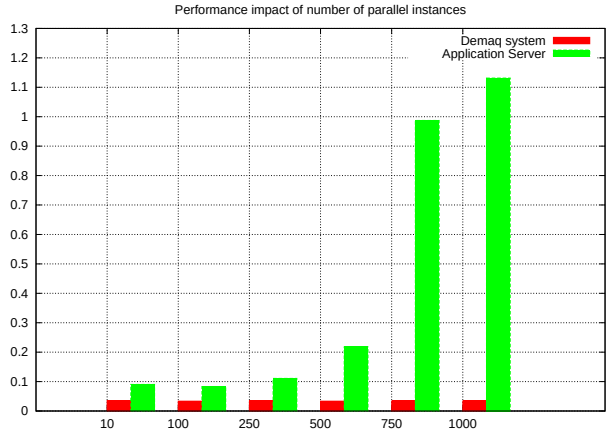

Figure 8: Impact of number of active instances

8. CONCLUSION AND FUTURE WORK

We propose a new programming model for distributed applications based on XML messaging. In our system, a declarative language that directly operates on messages and queues can be used to describe the processing logic in terms of message-driven rules, and application state is modeled exclusively using the message history. By treating application instance management as as a data management problem best addressed by a data management server, we get a fresh perspective on how to optimize the architecture of application servers. We extend the concept of data independence to the whole application stack, and introduce a message history-specific flavor of views, called *slicings*. Our rule compiler can then create optimized execution plans which correlate incoming requests with the relevant parts of the state by means of efficient access paths (such as indexes or materialized views). A result is improved scalability of our execution engine to large numbers of concurrent application instances: In particular, we can avoid loading and saving the complete application state from a database for every

processing step, which tends to take up a large fraction of conventional application servers' processing resources.

A brief performance evaluation of our runtime system confirms the potential of the proposed approach. It also illustrates the practical benefits of treating process instances as data in terms of scalability and performance.

Among others, an important direction of our future work is to further increase application scalability. In particular, we want to investigate how applications using our model can be *automatically* distributed to run on a network of processing nodes. This potentially includes vertical partitioning, where distinct parts of an application are run on different nodes, or horizontal partitioning, where data belonging to the same instance is processed by a particular instance.

Acknowledgments.

We thank Guido Moerkotte and Simone Seeger for their helpful comments on the manuscript. We also thank Balthasar Biedermann for his support with the implementation of the communication system, Dennis Knochenwefel for implementing the DB/2 integration, and Erich Marth for developing the visual editor.

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under grant MO 507/12-1.

9. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applictions*. Springer-Verlag, 2004.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager. In *SIGMOD Conference*, page 665, 2003.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[5] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A stream data management benchmark. In *VLDB*, pages 480–491, 2004.

[6] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[7] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0. Technical report, W3C, January 2007.

[8] S. Blanvalet. Managing a BPEL production environment. Technical report, Oracle Corporation, January 2006. `http://www.oracle.com/technology/pub/articles/bpel\_cookbook/blanvalet.html`.

[9] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. Technical report, W3C, January 2007.

[10] A. Böhm, C.-C. Kanne, and G. Moerkotte. Demaq: A foundation for declarative XML message processing. In *CIDR*, pages 33–43, 2007.

[11] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to XML repositories using active rules. *Computer Networks*, 39(5):645–660, 2002.

[12] D. Chamberlin, D. Florescu, J. Melton, J. Robie, and J. Siméon. XQuery Update Facility 1.0. Technical report, W3C, August 2007.

[13] D. D. Chamberlin, M. J. Carey, D. Florescu, D. Kossmann, and J. Robie. Programming with XQuery. In *XIME-P*, 2006.

[14] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[15] D. Engovatov. XML Query (XQuery) 1.1 requirements. Technical report, W3C, March 2007.

[16] M. F. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model (XDM). Technical report, W3C, January 2007.

[17] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a Native XML base management system. *VLDB Journal*, 11(4):292–314, 2003.

[18] D. Florescu, A. Grünhagen, and D. Kossmann. XL: a platform for Web Services. In *CIDR*, 2003.

[19] C. B. Foch. Oracle streams advanced queuing user's guide and reference, 10g release 2 (10.2), 2005.

[20] J. J. Garrett. Ajax: A new approach to web applications. Technical report, Adaptive Path, Feb. 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[21] J. Gray. Thesis: Queues are databases. In *Proceedings 7th High Performance Transaction Processing Workshop. Asilomar CA.*, 1995.

[22] IBM. WebSphere MQ, 2007. `http://www-306.ibm.com/software/integration/wmq/index.html`.

[23] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, pages 289–300, 2005.

[24] N. W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[25] M. Stonebraker. Too much middleware. *SIGMOD Record*, 31(1):97–106, 2002.

[26] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[27] M. Toyama. Parameterized view definition and recursive relations. In *ICDE*, pages 707–712. IEEE Computer Society, 1986.

[28] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.