AMetaCar

a Mediated eCommerce-Solution for the Used-Car-Market*

Lars Dittmann, Peter Fankhauser, and Andre Maric
GMD – German National Research Center for Information Technology
Integrated Publication an Information Systems Institute IPSI
Dolivostr. 15. 64293 Darmstadt, Germany
fankhaus@darmstadt.gmd.de, {Lars, Andre}@provirtual.de

Abstract

Providing integrated access to multiple information offerings on the Web poses some fundamentally new challenges for practical federated information systems. Other than conventional databases with an explicit and rather stable schema and expressive query APIs, Web-sources deliver semi-structured data without explicit logical structure and semantics, and have rather limited query capabilities. In this paper we present AMetaCar, a federated Web-information system, that provides integrated access to a number of existing Web-catalogues for used cars. At the wrapper level, the sources are continually monitored, and their implicit structure is explicated by means of XML. At the mediator level a hybrid approach is used, which combines a relational DBMS for the regular and common attributes of used-car offers with a persistent XML-DOM (Document Object Model) implementation for the irregular and heterogeneous attributes. XSL is used for presenting XML results. This overall approach combines the scalability of relational databases with the flexibility of XML for creating scalable and maintainable Information Brokering solutions.

* This work has been partially funded by the ESPRIT-project MIRO-Web (EP 25208).

1 Introduction

AMetaCar is a meta search engine for used cars. It accesses multiple autonomous and heterogeneous used car markets on the Web, extracts and homogenizes relevant information, continually materializes an integrated view, and thereby provides integrated access with enriched query facilities. On this basis, requests for used cars can be routed to multiple sites, providing a better recall, and due to the improved query capabilities also better precision. The actual deals are prepared by providing the direct contact to the original source. Thereby, AMetaCar operates as an e-commerce mediator.

AMetaCar can be regarded as a third generation meta search engine. The first generation, such as MetaCrawler [MC] works on regular search engines (like Altavista, Yahoo, etc.). These can be characterized by providing only one search field and performing little post-processing of results. The second generation, such asJango [PDE+97, Jan] or Acses.com [Acs] provides for more differentiated queries, typically for multiple product catalogues. An example is a best price search for books. The user enters the ISBN, title, or some other typically unique identification of the book and receives a list of possible seller orders by their price. But this approach is of limited applicability to catalogues of used cars. Whereas a book of one edition is easily comparable among multiple sites, based on only a few fixed attributes, used cars are and can not be described and compared as homogeneously. For example, two-year-old Volkswagen Golfs are likely to have large differences, and their descriptions when derived from multiple sources may also largely differ.

AMetaCar realizes a federated database architecture [She91], which tackles these problems as follows. Using the wrapper generation toolkit JEDI (Java Extraction and Dissemination of Information) [HFA+98] it extracts and partially homogenizes the heterogeneous individual offerings by transforming them to XML [XML98]. The extracted information is stored in a materialized view using a combination of a relational database with an XML storage engine [FGL+98, XQL99]. Queries are processed by loosely coupling the SQL query processor with an XML query processor. This approach allows to efficiently query regular and common attributes via relations, while maintaining the flexibility ofschemaless XML for irreconcilable structural and semantic differences. Queries result in XML-documents, which are presented by using XSL [XSL99].

The remainder of this paper is organized as follows. In Section 2, we detail problem domain and goals, and outline our overall approach along AMetaCar's architecture, which consists of three layers for wrapping, mediation, and presentation. In Section 3, we introduce the wrapping layer, which extracts information from sources and monitors sources for changes. In Section 4, we introduce the mediation layer, which materializes an integrated view in a hybrid relational and XML database. In Section 5, we describe the presentation layer. Finally, in Section 6, we summarize our results, and outline future work.

2 Overview

2.1 Goals and Overall Approach

AMetaCar combines the offers from a number of autonomous and heterogeneous Web-sources. Due to their autonomy the sources are free to change their offers and interface any time. In addition, the sources are heterogeneous. We can distinguish two kinds of heterogeneity:

Inter-source heterogeneity comprises syntactic and semantic deviations between the different sources. Sources organize the offers at different levels of granularity with respect to document structure and layout as well as to site structure. The layout-oriented HTML markup does rarely convey semantically meaningful correspondences between different offers. Furthermore, the sources provide different query capabilities supporting rich queries using diverse attributes of car offers. Not all these differences can be reasonably reconciled by mapping car-offers to a predefined, fixed database schema.

Intra-source heterogeneity arises from differences between offers from a single source. Depending on the potential seller, on the brand and other aspects, some attributes of individual offers may be missing, be represented at different levels of structural detail, and with different layout.

2.2 Architecture

The overall architecture is organized in three layers [Wie96] (Fig 1).

Wrappers access each individual source according to its capabilities and transform their data to a canonical datamodel. They are built with the help of the wrapper generation framework JEDI, which provides a library for sending queries and navigation requests to sites, and for transforming unstructured and semi-structured results to a structured object representation. Due to unreliable and rather slow connections to the individual sources the wrappers use a combination of complete materialization with monitoring of changes of the underlying sources. Due to the inter-source heterogeneity, the canonical data model combines a relational model for the stable and consistent parts of car-offers, and wellformed (type-less) XML for representing the irregularly structured parts of car offers.

The mediator loosely couples a relational database with a storage engine for wellformed XML documents. Queries are processed as follows: First, the SQL-part is evaluated for the relational portions by the SQL-Engine, which returns a set of records including the XML portions. From these records a persistent DOM (Document Object Model) representation (PDOM) is dynamically created, to further evaluate the XML-specific part of the query, which is expressed in XQL [XQL98], a query language specifically designed for XML. The remaining car-offers are randomly sorted by the shuffler to achieve a fair distribution of client contacts among the individual Web-sources.

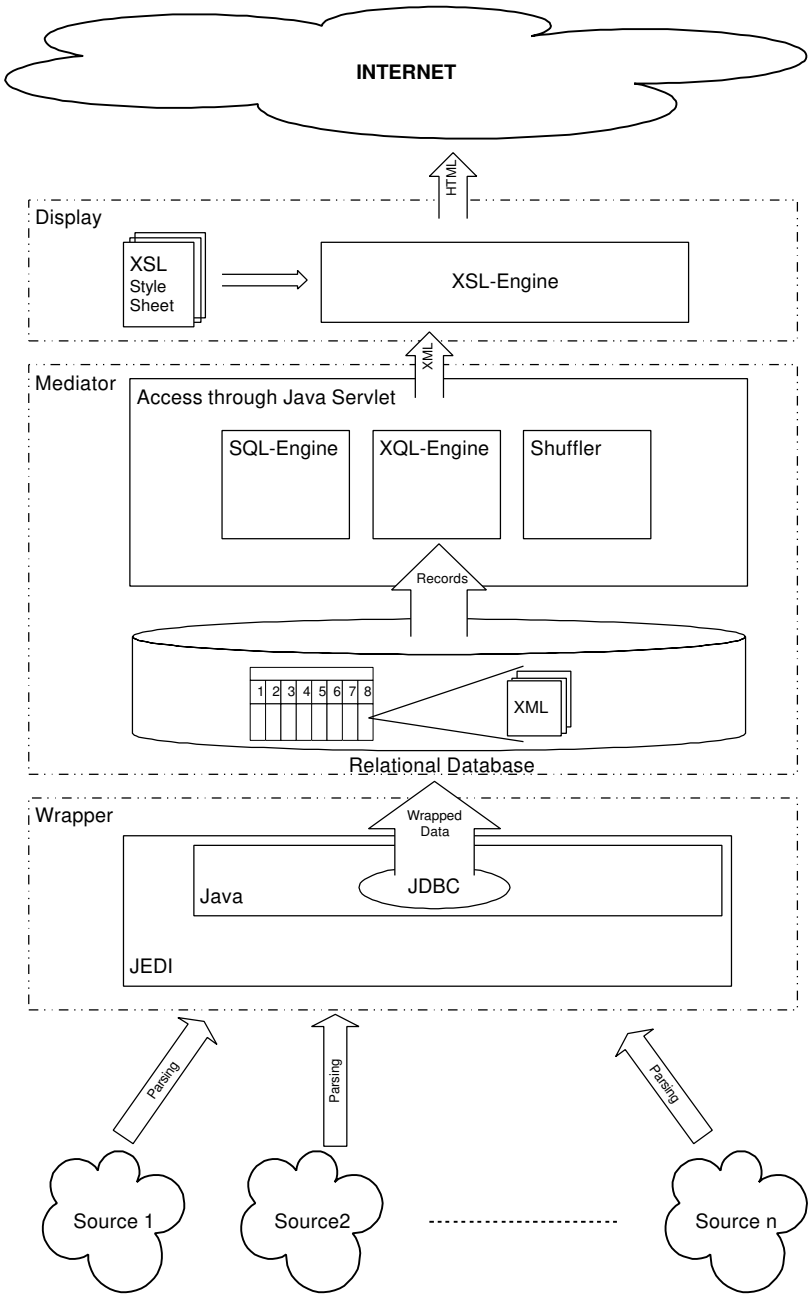

Figure 1: Architecture-Overview

The display layer is realized by means of XSL. An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is presented, by transforming it into an HTML-document. If the client's browser is capable of understanding XSL the formatting will be passed on to the client. Otherwise it will be done by a server-sided XSL-engine.

3 Wrapping

3.1 Source Interfaces

The AMetaCar Wrappers access specific Web-based used car markets and transform their data to a canonical datamodel to realize an integrated materialized view. The

sources we currently work on are Autobild [SRC1], Auto Scout 24 [SRC2], Faircar [SCR3], Mobile [SRC4] and Webauto [SRC5]. Figure 2 shows a simplified source.

The site is accessed by queries and navigation along three stages.

The *first stage* provides a page with a search mask. This page is not created dynamically, so it can be accessed directly by its URL. The example page contains several search criteria, such as Brand, Model, Price, KM and Year of Manufacture. The search value can be typically chosen from a set of alternatives presented in a list-box, radio buttons or checkboxes. The alternatives can be individual brands, price classes, or boundary values such as kilometers consumed. In addition, some sources also provide a form of free text search.

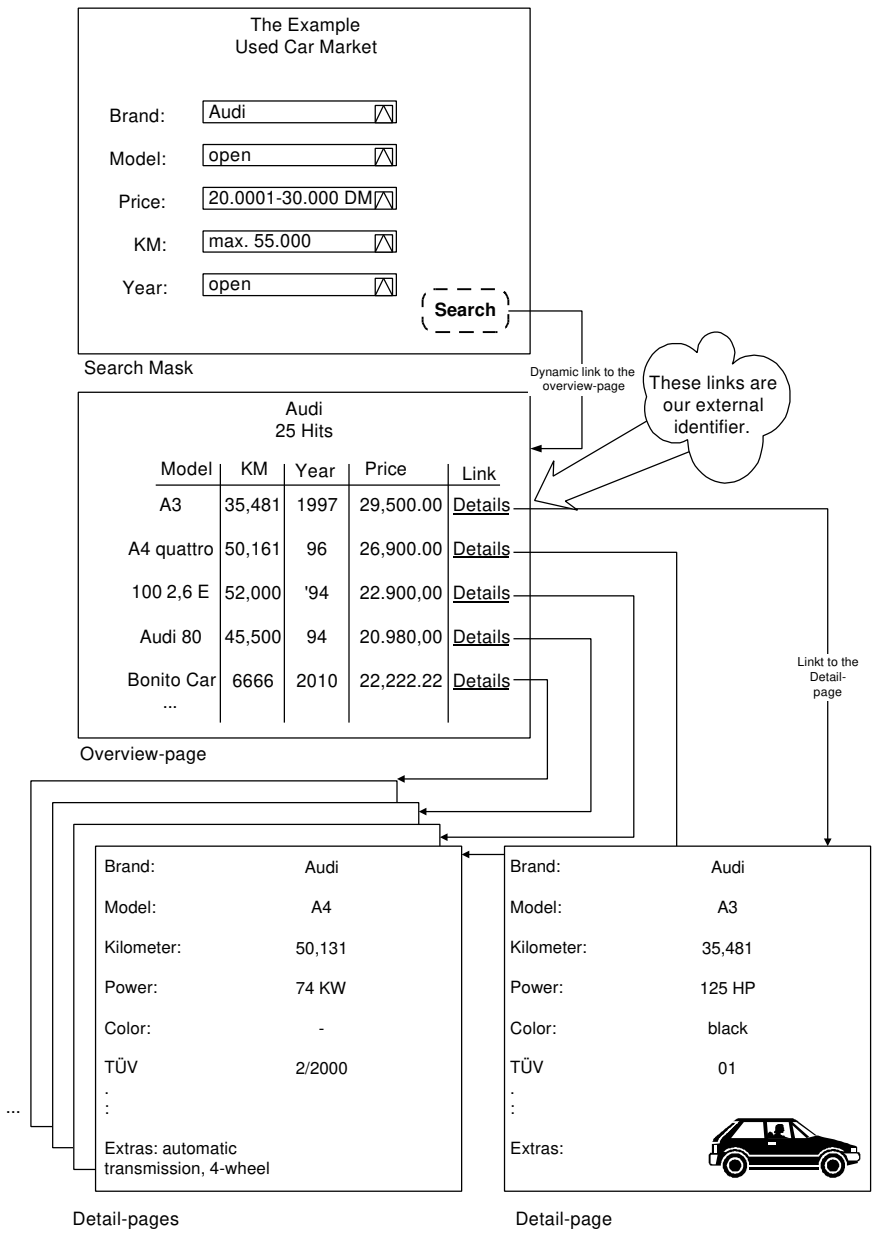

Figure 2: Navigating through a simplified source

A filled out search form leads to the *second stage*, which delivers overview pages with summary information about car offers and a link to the detailed information. The pages are created dynamically and some sources use cookies to maintain the context of a user when modifying or refining a search request. Thus they can only be accessed via the first stage.

The referred detail pages constitute the *third stage*. The detail-page is a static or dynamically created webpage. But for the lifetime of an offer its URL always provides a unique identifier. When the offer expires, the link is dead. This uniqueness of URLs allows monitoring changes in sources (see Section 3.3).

3.2 Extracting Structure from Semi-Structured Data

Whereas the pages of the first and the second stage have rather simple and consistent structure, the detail pages are semi-structured [Abi97]. Even within one source, individual car offers have a number of syntactic, structural and semantic differences, which makes it difficult to query and process them by automated means:

- *Unknown structure:* Although the data is typically stored in a relational database at some level of structural granularity, sources only provide some HTML layout, from which the logical structure needs to be explicated.
- *Irregular structure:* Depending on brand or type, and sometimes simply due to missing information, offers have different sets of attributes.
- *Implicit structure and semantics:* Besides a table of common attributes for a used car like Brand, model, year of manufacture, we find additional information, in the form of a text block consisting of a list of expressions. The meaning of the expressions is implicit. For example, the expression 4-wheel does not only mean that the car has four wheels, but also that all four wheels are used for powering the car.
- *Type and scaling inconsistencies:* Sometimes the power of a motor is given in horsepower, sometimes in kilowatt. Some other attributes such as production date may be at different level of detail, such as month and year, or year only.
- *Formatting inconsistencies:* Some attributes use different formats for their values: For example, the date of the next motor vehicle inspection (TÜV in German) may be represented by "2/2001", "02-2001", or even only "01".
- *Unconstrained values:* Some of our sources allow any Web user to enter their own used car offers, using unconstrained input masks. This can lead to meaningless attribute values, such as "Bonito Car" for car model.

To extract the implicit structure and semantics, and to overcome some of the inconsistencies we use JEDI (Java Extraction and Dissemination of Information), a wrapper generation framework developed at GMD-IPSI [HFA+98]. Its two most important components for our purposes are a fault-tolerant parser to extract structure and a self-describing, extensible object-model to flexibly represent the extracted structure.

The fault-tolerant parser uses attributed context free grammars for describing the implicit structure of sources. It employs a parsing strategy that uses potentially unlimited lookahead combined with a disambiguation scheme based on the specificity of multiple possible parsing solutions:

Consider, for example, the following simplified grammar that describes the coloring scheme used in a particular source:

```
.*'metallic'?(.*|' blue'|' red'| ' paint')*','.*
```

The leading and trailing `.*` are used to skip irrelevant portions, likewise the `.*` in the list of color and paint alternatives is used to skip irregular or irrelevant descriptions.
This grammar can be applied to the following lines:

```
metallic paint red
metallic blue
metallic yellow
```

For the first line the set of possible interpretations are:

```
.*
'metallic'.*
'metallic'.*' red'
'metallic'' paint'.*
'metallic'' paint'' red'
```

The first interpretation is the most generic interpretation. It groups everything into one .*. Also the subsequent lines are rather generic. The fault-tolerant parser of JEDI uses a ranking scheme that maximizes specificity, which roughly minimizes the number of wild-cards needed for an interpretation.

The self-describing object-model of JEDI is used to flexibly deal with the implicit semantic differences of such variations. For example, to describe the schema for the parsing results above, it is not necessary to explicitly enumerate paint, color, and other possible fillers by means of a union type. Rather, the generic type "coloring scheme" can take a number of self-describing types, which can easily be extended when new coloring aspects are required.

Below, we summarize how and to what extent JEDI can deal with the semi-structured car offerings along the dimensions introduced above.

- *Unknown structure* is explicated by means of grammars. When specifying the extraction rules, the fault-tolerance of JEDI allows focusing only on the structural aspects needed, while skipping irrelevant portions automatically. For utilizing HTML-layout structure, JEDI includes a special generic parser for HTML.
- *Irregular structure*: Ambiguous grammars allow to provide generic rules and optional rules to deal with structural variations, without losing the most specific interpretation.
- *Implicit semantics and unconstrained values*: This can not be solved by a parser alone, but requires an extensive thesaurus, for example, in the form of parser rules. When the set of possible interpretations is small, we do explicate the meaning, otherwise leave such portions unchanged, providing for free text search only.

- *Type and scaling inconsistencies*: The self-describing, extensible object model can deal with typing variations and JEDI's scripting facilities can be used to overcome scaling differences.
- *Formatting inconsistencies* can also be handled by ambiguous grammars. For example, variations like "2/2001" vs. "02-2001" need typically not to enumerate all possible variations, but can be interpreted by a grammar like .*([1-2][0-9])?.*[0-9]*. Provided that the syntactic context of date is clear, this can even deal with variations such as "february 2001", from which it determines at least the year.

3.3 Monitoring Changes

Fully scaled, AMetaCar will provide access to approximately 250.000 used car offers, an average of 25.000 offers from ten different sources. That means for every single source visiting at least 25.000 detail-pages and 1000 overview-pages to extract the external IDs. If one webpage is approximately 10 Kbytes large, a complete download will produce 260 Mbytes of traffic. With the added costs of reparsing all sources and rebuilding the database completely, the costs for a repetitive complete download are prohibitive. On the other hand, directly propagating user searches to the individual sites will result in bad performance due to unavailability of sites, the processing of results, and the rather heterogeneous query capabilities of the individual sites. Thus, even partially fetching data only on demand as described in [MW97] is not an option in our context.

The compromise is to completely download data once, materialize it at the mediator level (see next section), and then incrementally monitor for changes in the underlying sources, which can be regarded as a very simple form of a derived materialized view [Rou97]. For used car sources a complete download can typically be achieved by retrieving all available car brands form the respective list box, and posing a query for each brand.

Apart from schematic changes or changes in the layout, which typically require to manually upgrade the wrapper specification, two sorts of the much more frequent changes at instance level can be distinguished:

- New entries in the databases of our sources.
- Expired offers in the sources' database, which are still stored in our materialized view. This case is more critical, because the user might notice and complain about it more likely.

The most straightforward approach would be to catch these changes by explicit notification from the sources, realized, for example, by some regular push-service. However, all sources only provide for conventional pull by explicit search.

Thus we emulate change notification at the wrapper level with a regularly invoked monitor as follows. Using a search criterion, such as all brands chosen from the explicit set of alternatives in the list box, we produce a set of overview pages comprising all or at least the vast majority of current offers. These overview pages contain the URLs of the detail-pages, which constitute a unique identifier (see Figure 2). When comparing these identifiers to the corresponding identifiers of car offers in

the materialized view, three cases need to be distinguished. (1) The identifier is not yet available in the materialized view, meaning a new offer has been detected. (2) The identifier is only available in the materialized view, meaning the offer has expired. (3) The identifier is available in both, meaning the offer has not yet expired. The monitoring algorithm is described in more detail below.

Monitor:

```
For each source do:
   DatabaseIDs = all IDs available in
        materialized view
   For each car-brand do:
      1. SourceIDs = Union of all new IDs extracted from
         the overview pages
      2. For all IDs in SourceIDs - DatabaseIDs do:
         fetch new offer for ID from source, and insert
         it in the materialized view
      3. For all IDs in DatabaseIDs - SourceIDs do:
         delete expired offer for ID
      4. For all IDs in DatabaseIDs intersect SourceIDs
         do:
         update timestamp of offer for ID in
         materialized view.
```

Our internal timestamp of offers is used for scheduling the monitoring process and possibly for ranking the car offers according to recency. The monitoring process always prioritizes the sources and car-brands with the oldest timestamp for a particular source and car brand. This allows distributing the monitoring load evenly among sources and car brands. We intend to experiment also with more refined strategies, such as prioritizing monitoring requests according to use-statistics, for example, by prioritizing monitoring requests for the most often requested car-brands. Note that this overall approach will not catch changes in attribute values, such as price, when they are not reflected by a new URL. For a more comprehensive and general treatment of change monitoring see, for example, the CQ (Continual Querying Approach) in [LPT+98, LPT99, PL98].

4 Mediating

4.1 Hybrid data-model

At the mediation level car offers from various sources are merged into a single materialized view. While car offers of individual sources depict some commonalties, such as car-brand, model, and price, they also show a great deal of variety with respect to level of detail, such as special legal data. It is impractical to force all these variations into a single, uniform global schema.

One extreme of modeling such heterogeneous data is to dispose of a schema entirely by using wellformed XML. But querying schemaless XML-documents is not as efficient as querying a relational database, both, from the system perspective, as well as from the user perspective. The other extreme is to design a complete global schema, and derive a fully normalized relational database design from it. But this

results in a lot of small tables for all irregularities involved or has to make extensive use of null-values or has to discard the irregularities.

Thus we adopt a *hybrid* approach, along the lines described in [LAW99]. For the regular portions we use the relational database MySQL [MySQL]. The irregular portions are put into an XML-document, which is stored as a CLOB (binary large object).

Here is a simple example record.

Internal ID	Source	External ID	Model	KM	Year	…	Auto Trans	XML
234	Webauto	http//:...	A4	1000	98	…	False	<techdata> <gears> 4 </gears> … </techdata> <appearancedata> <color> black </color> … </appearancedata> …

The XML-portion typically forms a tree with depth 2. The top levelmodels categories of attributes such as technical data vs. appearance data, the next level contains the actual attributes represented by XML-elements. However, these attributes need not be atomic. For example, when a car offer provides purchase-contacts in highly structured fashion, this will form an accordingly further structured attribute. Also no other restrictions are imposed; categories and attributes may be missing. The structure is not fixed. It can change and evolve as the user preferences change and evolve.

The added flexibility comes of course at a cost. Firstly, XML in its serialized form with full markup requires unnecessary storage. However, standard compression techniques apply very well to XML-documents with the redundancy induced by their markup. Secondly, querying XML documents is not very efficient, unless extensive indices are created and maintained. Therefore, guided by the main sorts of queries we want to support, we also put important, but irregular search-criteria, such as automatic transmission, as extra boolean attributes into the relational portion. These attributes are abstractions from the more detailed information available in the XML portion. For example, for automatic transmission, the XML portion may still differentiate between a tip tronic, 3 gear or 4 gear automatic transmission etc. in a <transmission> element. This redundancy makes the query faster by taking advantage of the SQL engine.

4.2 Querying

The mediator is realized as a JAVA servlet, comprising several components. For querying the relational portions we use the SQL-Engine implemented inMySQL. For querying the XML portions, we use XQL, a query language for XML [XQL98], implemented in the GMD-IPSI XQL-Engine [XQL99], which implements a semi-

structured database [GMW99, MW97] for XML. Thus we need to combine SQL- and XQL-queries. Lacking a stable and scalable general-purpose query processor capable of dealing with a mixture of SQL and XQL, we adopt a loosely coupled, sequential approach.

With this, query processing involves the following steps, as shown in Figure 3:

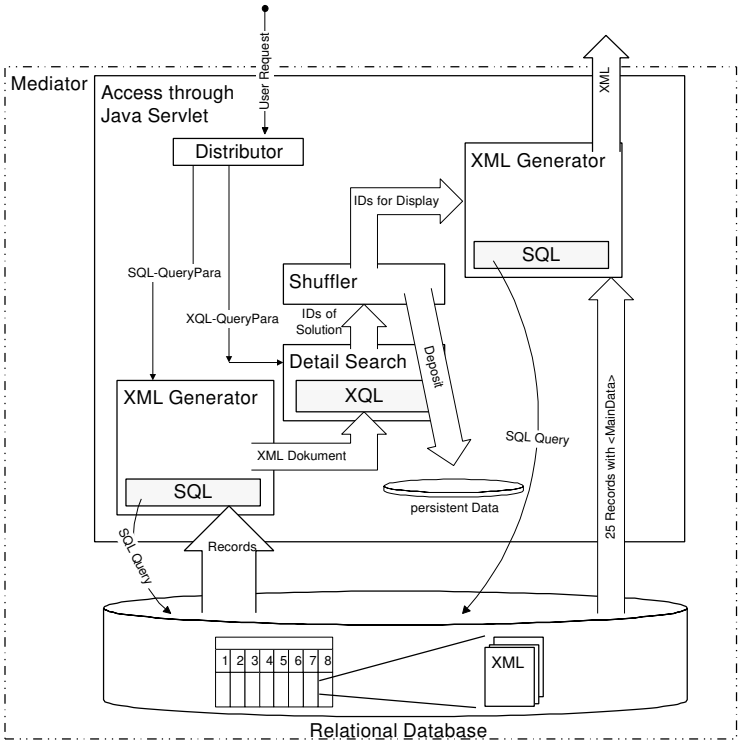

Figure 3: Query-mechanism

1. The distributor receives a user query, and splits it into its SQL-part and its XQL-part
2. The SQL part is forwarded to the XML-generator, and the XQL-part is forwarded to the detailed search module.
3. The XML-generator sends the SQL-query to the database and receives the corresponding internal IDs and XML-documents.
4. From the IDs the generator creates an XML-document and passes it to the detail search module. This module creates on the fly a DOM [DOM98] representation of the XML document for processing the query in the next step.
5. The detail search module sends the XQL-query to the XQL-Engine and receives a set of ID's matching the additional criteria.
6. The IDs are passed on to the shuffler.
7. The shuffler picks 25 of them randomly and sends theirs IDs to the second XML generator (see Section 4.3). It might be a large number of possible solutions, which can not be demonstrated at the same time. All results are kept in the form of internal IDs in a persistent storage of the servlet, to allow for quickly browsing back and forth through several sets of 25 hits.
8. The second XML generator runs a second SQL-query on the database, requesting the complete records for the chosen ID's. The results are transformed into an

XML-file and sent to the display module. Thus, only at this level the complete car information needs to be fetched from the database.

For certain cases this approach may lead to performance problems. In the worst-case, the SQL portion of the query retrieves the entire database (approximately 200 MB). This happens, for example, when a user searches for any car with a hydraulic ramp for handicapped persons, which is not represented as a relational attribute. The XQL-engine must parse and generate a persistent DOM representation for the entire irregular portion of the database, before it can run the actual query. This makes the size of the XML-document the performance bottleneck of the query processor.

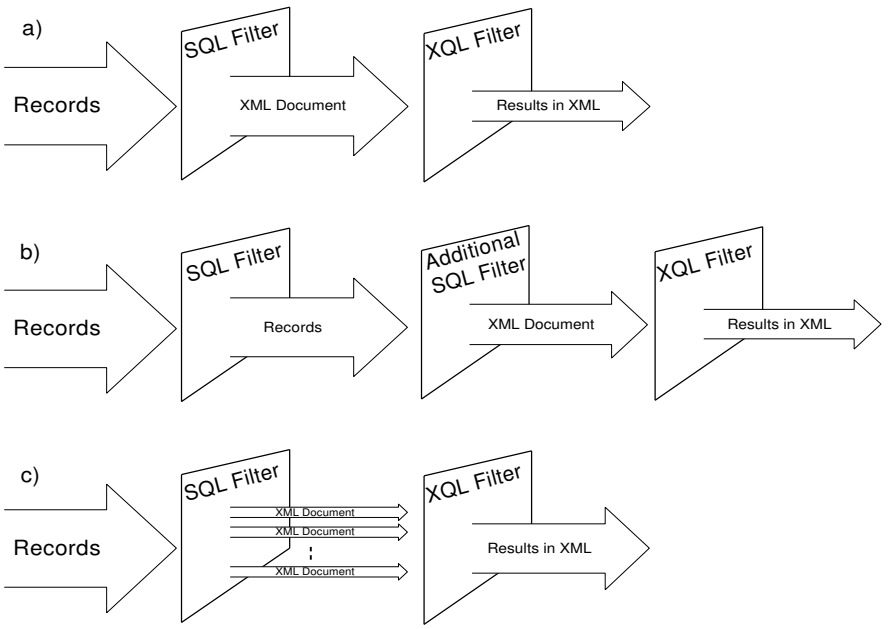

Figure 4: Queries as Filters

There are several ways to avoid such bottlenecks. The two query engines can be regarded as filters, applied subsequently to the relational database. With this perspective, optimization can be achieved by either introducing additional filters or by splitting up and partially deferring the costly generation of the XML-portion.

Figure 4 illustrates these two approaches in more detail:

1. *Introducing additional SQL-Filters (Figure 4 b).*
   We concentrate on two methods for this approach:
   - The first SQL-query performs a string search on the XML-data field to pre-select records. This can be a search for a tag name or for PCDATA in an element. This does not make the XQL-query obsolete. Example: The user searches a car with more than 2000 cm^2 of cubic capacity. Now we can start a string search for "Hubraum" (German for cubic capacity) and any number greater than 2000. But we do not know, whether the found documents match our criteria. For example, the string search will also find a car with 1600 cm^2 cubic capacity and tires with the name Pirelli 3000.
   - We can use more of the extra boolean-fields as described in Section 4.1.

2. *Incrementally processing smaller XML-documents (Figure 4 c).*
   Option 1 is not applicable for all those cases, where the search space can not be reasonably restricted with an SQL query. For such cases, one can also limit the size or number of records queried in one pass, and perform a number of passes, rather than producing one single XML Document for all records delivered by the SQL part. This can be particularly suitable, when the user wants to see a limited number of hits. As soon as this number is reached, the results can be presented to the user for further inspection, without iterating through the whole database.

4.3 Shuffling

Ideally, user queries are exact enough to deliver only a few results, or the results can be ranked meaningfully. This is however rarely the case. User queries tend to be vague, and possible ranking criteria, such as price, tend to be rather invariant for a particular brand, model, and year. When multiple attributes are taken into account for ranking, potentially complex and unintuitive aggregation functions would be needed for ranking. Here is an example where it is not possible to objectively determine which car-offer is best, all other attributes considered equal.

	Price in DM	Kilometers
Car No. 1	5000	100.000
Car No. 2	4500	120.000

Asking the user might be a solution but not all users are willing to fill out another form describing their preferences. If we would use our own set of ranking criteria, a smart car-dealer could find out our criteria and modify his entry in a way that it always appears first in the list. But also for the sake of fairness for the car dealers and individual original sources, we wish to distribute client-dealer contacts to evenly.

These observations lead us to randomly shuffle the results. To this end we use a two step approach.

First, we sort the contents of the car offers in the database randomly such that the several providers have equal chance to match user requests averaged over all possible user requests. For a particular user request, however, the results can still be biased according to the actual distribution of car-offers from the several providers. For example, a source that specializes on a particular brand, and therefore has significantly more offers for this brand than others, has a better chance to be represented in the result set with larger probability values.

To avoid this the shuffler inserts each retrieved car offer only with a probability smaller than 1 as long as there are enough additional offers available. By choosing source specific probabilities, the biases can be evened out to a certain extent. There are two approaches to accomplish this. One approach is to maintain statistics for certain simple queries, such as brand, which store the relative number of available offers per attribute value and source. With this information, user queries can be analyzed to choose an individual probability per source. The other approach is to decrease the selection probabilities incrementally per source, for every offer taken from the source. This has the advantage that it needs no separate statistics, and can

achieve an unbiased distribution for arbitrary queries. The disadvantage is that usually more offers need to be processed.

For queries delivering only a few results or results that can be meaningfully ranked, we intend to make shuffling optional.

5 Display

The results of a user's request are returned in the form of XML-documents of variable size. Because we intend to serve mainly casual users with rather different demands, we create these pages completely on the fly. Another option would be to materialize some portions and reuse common portions for several users, as described in [LR99].

To render the XML documents with arbitrary WebBrowsers, we use XSL, a language that allows attaching layout semantics for a class of XML-documents. In our case, XSL-stylesheets determine how an instance of the class is transformed into an HTML-document. Our XML results have only at the top level a regular structure. Thus, they do not entirely conform to a DTD. XSL provides for flexible means to deal with such irregular structures, in particular by performatives that can apply matching templates to arbitrary depth of a document.

Depending on the browser capabilities of a user, the XSL-engine is server-sided to create HTML, or the XML document together with the XSL-stylesheet is delegated to the client.

6 Conclusions and further research

AMetaCar realizes a federated information system for the used car market. It tackles the main dimensions of federated information system – heterogeneity, distribution, and autonomy - as follows. (a) The implicit structure of heterogeneous sources is explicated by means of XML. Only the common model portions of sources are mapped to an integrated schema, which is implemented by a relational database. Irreconcilable portions are represented as wellformed (schemaless) XML. This allows to efficiently query the common portions with SQL, and to query source specific structures with XQL. (b) Distribution is dealt with materialization combined with a continual monitoring mechanism. Due to unreliability of Internet connections, and the practically limited API interfaces of sources, this approach is more suitable than propagating portions of queries to the underlying sources on demand. (c) Autonomy of sources is attacked in a twofold way. Actual deals between clients and providers are delegated to the original sources in order not to interfere with their individual business models. Furthermore, special measures are taken to distribute client – provider contacts evenly.

At the time of writing this paper, we have finalized the wrappers for five sources, and are gaining first experiences with the hybrid mediator implementation. Technically, we will focus further work to the following issues. (a) To allow for more flexible and more general query processing, we will investigate how to couple relational with XML storage more closely. Along these lines we consider to maintain XML-documents as BLOBs which contain small persistent DOMs, and investigate design

options to use XQL-expressions as an integral part of SQL queries, using similar concepts as described in [BAN+97]. (b) To ease the design and maintenance of wrappers and mediators we look into techniques for generating their specifications from example interactions on the individual sources.

Methodologically, the most pressing questions for mediated e-Commerce solutions such as AMetaCar, arise from a genuine interplay between heterogeneity and autonomy. We envisage the next generation of mediated e-Commerce to provide also for transparency of diverse business models, such as diverse payment and delivery schemes. At the information modeling side, this requires much more elaborate means to model and reason about heterogeneous product descriptions. At the process modeling side, this needs robust protocols to couple heterogeneous and autonomous business models.

7 References

[Abi97] Serge Abiteboul: Querying Semi-Structured Data.*ICDT 1997*; pp.1-18.

[Acs] Acses: http://www.acses.com/

[BAN+97] Klemens Böhm, Karl Aberer, Erich J. Neuhold, Xiaoya Yang: Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM, *VLDB Journal, Volume 6, Number 4, November 1997*; pp. 296-311.

[DOM98] Document Object Model (DOM) Level 1 Specification, Version 1.0, W3C Recommendation 1 October, 1998, http://www.w3.org/TR/REC-DOM-Level-1/

[FGL+98] Peter Fankhauser, Georges Gardarin, M. Lopez, J. Munoz, A. Tomasic: Experiences in Federating Databases: From IRO-DB to MIRO-Web. *Ashish Gupta, Oded Shmueli, Jennifer Widom (Eds.): Proceedings of the 24th annual International Conference on Very Large Data Bases, VLDB´98, August 24-27,1998*; pp.655-658.

[GMW99] Roy Goldman, Jason McHugh, Jennifer Widom: From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. *WebDB (Informal Proceedings) 1999*; pp. 25-30.

[HFA+98] Gerald Huck, Peter Fankhauser, Karl Aberer, Erich J. Neuhold: Jedi: Extracting and Synthesizing Information from the Web. *Michael Halper (Ed.), Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, CoopIS'98, August 20-22, 1998*; pp. 32-43.

[Jan] Jango: http://www.jango.com/ or http://www.shopbot.com/

[LAW99] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating Structured and Semistructured Data. *To appear in Proceedings of the Seventh International Conference on Database Programming Languages, Kinloch Rannoch, Scotland, September 1999.*

[LPT+98] Ling Liu, Calton Pu, Wei Tang, David Buttler, John Biggs, Tong Zhou, Paul Benninghoff, Wei Han, Fenghua Yu: CQ: A Personalized Update

Monitoring Toolkit. *SIGMOD Conference 1998;* pp. 547-549.

[LPT99] Ling Liu, Calton Pu, Wei Tang: Continual Queries for Internet Scale Event-Driven Information Delivery. *To appear in Special issue on Web Technologies, IEEE Transactions on Knowledge and Data Engineering, Jan. 1999.*

[LR99] Alexandros Labrinidis, Nick Roussopoulos: On the Materialization of WebViews. *WebDB (Informal Proceedings) 1999;* pp. 79-84

[MC] Metacrawler: http://www.metacrawler.com/

[MW97] J. McHugh and J. Widom. Integrating Dynamically-Fetched External Information into a DBMS for Semistructured Data. *SIGMOD Record, 26(4), December 1997;* pp. 24-31

[MW97] J. McHugh and J. Widom. Query Optimization for Semistructured Data. *Technical Report, November 1997.*

[MySQL] MySQL Homepage: http://www.mysql.com/

[PDE+97] Mike Perkowitz, Robert B. Doorenbos, Oren Etzioni, Daniel S. Weld: Learning to Understand Information on the Internet: An Example-Based Approach, *Journal of Intelligent Information Systems, Volume 8, Issue 2, March 1997;* pp. 133-153.

[PL98] Calton Pu and Ling Liu: Update Monitoring: The CQ project. *(invited paper) The 2nd International Conference on Worldwide Computing and Its Applications – WWCA'98, Tsukuba, Japan, Lecture Notes in Computer Science 1368;* pp. 396-411.

[Rou97] Nick Roussopoulos: Materialized Views and Data Warehouses. *KRDB 1997,* pp. 12.1-12.6.

[She91] Amit P. Sheth: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *VLDB 1991;* pp. 489-521.

[SRC1] Source1: Autobild, http://www.autobild.de/automarkt/ or http://www.autoboersedeutschland.de/

[SRC2] Source 2: Auto Scout 24, http://www.autoscout24.de/ or http://www.mastercar.de/

[SRC3] Source 3: Faircar, http://www.faircar.de/

[SRC4] Source 4: Mobile.de, http://www.mobile.de/

[SRC5] Source 5: Webauto, http://www.webauto.de/

[Wie96] Gio Wiederhold: Intelligent Integration of Information, Journal of Intelligent Information Systems, JIIS, 6(2/3): 93-98 (1996)

[XQL98] XML Query Language (XQL), proposal, September 1998, http://www.w3.org/TandS/QL/QL98/pp/xql.html/

[XQL99] Hompage of the GMD-IPSI XQL-Engine: http://xml.darmstadt.gmd.de/xql/index.html

[XSL99] Extensible Stylesheet Language (XSL) Specification, W3C Working Draft, 21 Apr 1999, http://www.w3.org/TR/WD-xsl/