ONE-TO-MANY DATA TRANSFORMATION OPERATIONS

Optimization and execution on an RDBMS

Paulo Carreira
Faculty of Sciences, University of Lisbon, C6 - Piso 3, 1749-016 Lisboa, Portugal
paulo.carreira@tagus.ist.utl.pt

Helena Galhardas, João Pereira and Andrzej Wichert
INESC-ID, Avenida Prof. Cavaco Silva, Tagus Park, 2780-990, Porto-Salvo, Portugal
hig@inesc-id.pt, joao@inesc-id.pt, wichert@inesc-id.pt

Keywords: Data Warehousing, Data Cleaning, Data Integration, ETL, Query optimization

Abstract: The optimization capabilities of RDBMSs make them attractive for executing data transformations that support ETL, data cleaning and integration activities. However, despite the fact that many useful data transformations can be expressed as relational queries, an important class of data transformations that produces several output tuples for a single input tuple cannot be expressed in that way.

To address this limitation a new operator, named *data mapper*, has been proposed as an extension of Relational Algebra for expressing one-to-many data transformations. In this paper we study the feasibility of implementing the mapper operator as a primitive operator on an RDBMS. Data transformations expressed as combinations of standard relational operators and mappers can be optimized resulting in interesting performance gains.

1 INTRODUCTION

The setup of modern information systems comprises a number of activities that rely, to a great extent, in the use of data transformations (Lomet and Rundensteiner, 1999). Well known cases are the migration of legacy data, the ETL processes that support data warehousing, data cleaning processes and the integration of data from multiple sources.

One natural way of expressing data transformations is to use a declarative query language and to specify the data transformations as queries (or views) over the source data. Because of the broad adoption of RDBMSs, such language is often SQL, a language based on Relational Algebra (RA). Unfortunately, due to its limited expressive power (Aho and Ullman, 1979), RA alone cannot be used to specify many important classes of data transformations.

An important class of data transformations that may not be expressible in RA are the so called one-to-many data transformations (Carreira et al., 2006), that are characterized by producing several output tuples for each input tuple. One-to-many data transformations occur normally due to the existence of *data heterogeneities*, i.e., due to the use of different representations, of the same data of source and target schemas (Rahm and Do, 2000). For instance, source data may consist of salaries aggregated by year, while the target data consists of salaries aggregated by month. Hence, each input row has to be converted into multiple output rows, one for each month. In this case, each input row corresponds to at most twelve output rows. However, expressing such data transformations as RA expressions is hampered by the fact such bound cannot always be established a-priori. Consider the following example:

Example 1.1 Consider the source relation `LOANS[ACCT, AM]` (represented in Figure 1) that stores the details of loans per account. Suppose that `LOANS` data must be transformed into `PAYMENTS[ACCTNO, AMOUNT, SEQNO]`, the target relation, according to the following requirements:

1. In the target relation, all the account num-

Relation LOANS

ACCT	AM
12	20.00
3456	140.00
901	250.00

Relation PAYMENTS

ACCTNO	AMOUNT	SEQNO
0012	20.00	1
3456	100.00	1
3456	40.00	2
0901	100.00	1
0901	100.00	2
0901	50.00	3

Figure 1: Illustration of an unbounded data-transformation. *(a)* The source relation LOANS on the left, and *(b)* the target relation PAYMENTS on the right.

bers are left padded with zeroes. Thus, the attribute ACCTNO is obtained by (left) concatenating zeroes to the value of ACCT.

2. The target system does not support payment amounts greater than 100. The attribute AMOUNT is obtained by breaking down the value of AM into multiple parcels with a maximum value of 100, in such a way that the sum of amounts for the same ACCTNO is equal to the source amount for the same account. Furthermore, the target field SEQNO is a sequence number for the parcel. This sequence number starts at 1 for each sequence of parcels of a given account.

The implementation of data transformations similar to those requested for producing the target relation PAYMENTS of Example 1.1 is challenging, since the number of output rows, for each input row, is determined by the value of the attribute AM. In this case, the upper bound on the number of output rows cannot be determined by analyzing the data transformation specification. We designate these data transformations as *unbounded* one-to-many data transformations. Other sources of unbounded data transformations exist like, for example, converting collection-valued attributes of SQL:1999 (Melton and Simon, 2002), where each element of the collection is mapped to a new row in the target table. In the context of data-cleaning, one commonplace transformation is converting a list of elements encoded as a string attribute.

Currently, one has to resort, either to a general purpose programming language, to some flavor of proprietary scripting of an ETL tool, or to an RDBMS using *recursive queries* of SQL:1999 (Melton and Simon, 2002), or some sort of *Persistent Stored Modules* (PSMs) (Garcia-Molina et al., 2002, Section 8.2) like stored procedures or *table functions* (Eisenberg et al., 2004).

To address the problem of expressing one-to-many data transformations in a declarative and optimizeable fashion, specialized relational operator named *mapper* was recently proposed as an extension to RA (Carreira et al., 2006). Informally, a mapper is applied to an input relation and produces an output relation. It iterates over each input tuple and generates one or more output tuples, by applying a set of domain-specific functions. This way, it supports the dynamic creation of tuples based on a source tuple contents.

Although mappers appear implicitly in systems supporting schema and data transformations underlying ETL, data cleaning and data warehousing (Galhardas et al., 2000; Raman and Hellerstein, 2001; Cui and Widom, 2001; Amer-Yahia and Cluet, 2004), as far as we know, their execution and optimization has never been properly studied.

This paper studies the feasibility of extending RDBMSs with the mapper operator. There are several reasons to do so: First, implementing the mapper operator as a relational operator opens interesting optimization opportunities since expressions that combine the mapper operator with standard RA operators can be optimized. Second, many data transformations are naturally expressed as relational expressions, leveraging the optimization strategies already implemented by RDBMSs (Chaudhuri, 1998). Third, such extension further equips RDBMSs for data transformation activities, broadening their applicability in a wider range of data management activities. We remark that our idea of using RDBMSs as data transformation engines is not revolutionary, see (Haas et al., 1999). Furthermore, several RDBMSs like Microsoft SQL Server and Oracle already include additional software packages specific for ETL tasks.

Our contributions are the following: *(i)* an SQL-like concrete syntax for the mapper operator accomplished by extending the select statement, *(ii)* the study of several query rewriting possibilities to be incorporated in the query optimizer and *(iii)* an experimental validation of the usefulness of implementing the mapper operator by comparing its physical implementation with alternative RDBMS solutions.

The rest of the paper is organized as follows. Section 2 introduces the mapper operator and exposes its concrete syntax by example. Then, in Section 3 we discuss how to extend the query optimizer to handle mappers. In Section 4, we report on a series of experiments to ascertain the feasi-

bility of implementing the mapper operator and finally Section 5 concludes.

2 THE MAPPER OPERATOR

The mapper operator is formalized as a unary operator μ_F that takes a relation instance of the source relation schema as input and produces a relation instance of the target relation schema as output. The operator is parameterized by a set F of functions, which we designate as *mapper functions*. The intuition is that each mapper function f_{A_i} expresses a part of the envisaged data transformation, focused on a non-empty set A_i of attributes of the target schema. A key insight is that, when applied to a tuple, a mapper function can produce a set of values in the domain of its target attributes $Dom(A_i)$, rather than a single value. Further details can be found in (Carreira et al., 2005b).

The mapper operator is formally defined as follows: Given a set of mapper functions $F = \{f_{A_1}, ..., f_{A_k}\}$, the *mapper* of a relation s with respect to F, denoted by $\mu_F(s)$, is the relation instance of the target relation schema defined by

$$\mu_F(s) \stackrel{\text{def}}{=} \{t \in Dom(Y) \mid \exists u \in s \text{ s.t. } t[A_i] \in f_{A_i}(u),\ \forall 1 \leq i \leq k\} \quad (1)$$

We can express the data transformation of Example 1.1 by means of a mapper $\mu_{acct,amt}$, comprising two mapper functions. The function *acct* is the mapper function that returns a singleton with the account number `ACCT` properly left padded with zeroes, while *amt* is a mapper function that produces the attributes `[AMOUNT,SEQNO]`, s.t., $amt(am)$ is given by $\{(100, i) \mid 1 \leq i \leq (am/100)\} \cup \{(am\%100, (am/100) + 1) \mid am\%100 \neq 0\}$, where % represents the modulus operation. For instance, if v is the source tuple $(\texttt{901}, \texttt{250.00})$, the result of evaluating $amt(v)$ is the set $\{(\texttt{100}, \texttt{1}), (\texttt{100}, \texttt{2}), (\texttt{50}, \texttt{3})\}$. Given a source relation s including v, the result of the expression $\mu_{acct,amt}(s)$ is another relation that contains the set of tuples $\{\langle\texttt{'0901'}, \texttt{100}, \texttt{1}\rangle, \langle\texttt{'0901'}, \texttt{100}, \texttt{2}\rangle, \langle\texttt{'0901'}, \texttt{50}, \texttt{3}\rangle\}$.

2.1 Concrete syntax

The mapper operator can be easily embedded into the SQL syntax by incorporating mapper functions as expressions into the *select* block. The main change consists of replacing the the standard list of columns and expressions that follow the **select** keyword with a list of mapper functions. A mapper function can be identified by an expression or, alternatively, by an inline specification of a mapper function. A particular kind of expression is a mapper function call. In this case, the function is specified outside the select statement using a more appropriate programming language. We note that a similar assumption occurs w.r.t. the computation of aggregates in SQL, in the sense that aggregate functions like e.g., `COUNT` or `SUM`, are implemented elsewhere and then embedded in the select statement as parameters of the aggregation operator. An inline mapper function is specified by a **map** clause followed by a list of attributes. These attributes are the mapper function's attributes that contribute to the schema of the output relation. The logic of the mapper function in this case is enclosed within a **begin end** block.

2.2 Filters

Filters are specified through the boolean expression of the **where** clause. Two kinds of filters can be specified, *(i) a-priori filters*, that apply to each tuple of the input relation, which are evaluated before the mapper and *(ii) a-posteriori filters* that are evaluated on the output of the mapper, which are used to limit the mapper results. They are identified by sub-expressions defined over particular sets of columns. Sub-expressions that are defined only over the columns of the input relation define a-priori filters, while sub-expressions that are defined over columns generated by the mapper functions define a-posteriori filters.

3 OPTIMIZATION

While parsing a *select* block, as soon as mapper function is found, the parser knows that a mapper operator is present. In this case, upon parsing the query successfully, the parser identifies all the mapper functions being used and computes the output schema of the mapper. The input schema of the mapper is determined by the schema of input relations. Once the input schema is known, the input columns specified for each mapper function can be validated. The following step is to rewrite the filter condition into the conjunctive normal form and validate it considering the input and output schemas. Then, each con-

junct is analyzed to decide whether it constitutes a candidate to an a-priori or to an a-posteriori filter. In the query presented in Figure 2, the sub-expression `ACCOUNTS.STATUS = 'O'` defines an a-priori filter while the sub-expression `AMOUNT < 50` defines an a-posteriori filter.

```
1: select map acct(ACCT) as ACCTNO,
2:        map amt(AM) as AMOUNT, SEQNO
3: from LOANS, ACCOUNTS
4: where ACCOUNTS.ACCTN = LOANS.ACCT
5:   and ACCOUNTS.STATUS = 'O'
6:   and AMOUNT < 50
```

Figure 2: A query that selects small payments of open accounts by implementing a mapper together with a-priori and a-posteri filters.

We note also that in some situations it is not possible to clearly separate these two kinds of filters. For example, if the condition is dependent on both input and output columns of the mapper like e.g., `AMOUNT < ACCOUNTS.WDRAWLIMIT`, where `AMOUNT` is an output attribute produced by a mapper function and `ACCOUNTS.WDRAWLIMIT` is an attribute of the input relation. In these cases, the predicate can only be evaluated after all the mapper functions, i.e., a-posteriori.

The specification of a-posteriori filters in the **where** clause opens an interesting possibility of defining the condition using mapper functions. The sets of values returned by mapper functions can be tested with set operators like **in** or **exists**.

Moreover, whenever the input relation is defined through join operations, some of the conjuncts can be immediately pushed down into the appropriate join operators. Generically, the query plan that results from this process applies an a-posteriori filter to a mapper operator. This mapper operator, in turn, is evaluated over the input relation resulting from applying an a-priori filter to a query sub-plan that represents the input relation. This concept is illustrated in Figure 3. Therein, the filter $\sigma_{\texttt{AMOUNT < 50}}$ is applied to the mapper $\mu_{acct,amt}$ which takes as input the tuples of $\texttt{ACCOUNTS} \bowtie_{\texttt{ACCOUNTS.ACCTN=LOANS.ACCT}} \texttt{LOANS}$ that are not filtered by $\sigma_{\texttt{ACCOUNT.STATUS = 'O'}}$. The plans so obtained are then handed to the query optimizer where they undergo a series of rewritings that turn them into equivalent ones that are more efficient to evaluate. Besides the usual rewritings implemented by RDBMSs, others, specific to mappers can be introduced. Some of these rewritings are interesting because they take direct advantage of the mapper semantics. Herein, due to space limitations, we briefly sketch the main

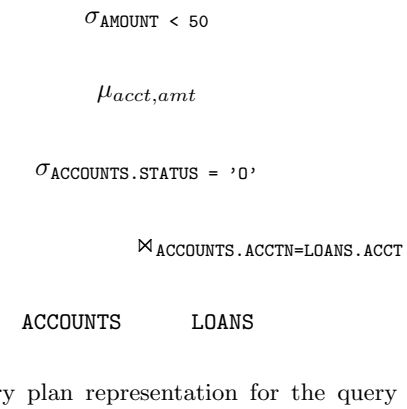

Figure 3: Query plan representation for the query presented in Figure 2.

ideas. Please refer to (Carreira et al., 2005b; Carreira et al., 2006) for details about rewriting rules and their corresponding proofs of correctness.

Projections A projection applied to a mapper is an expression of the form $\pi_Z(\mu_F(s))$. Those mapper functions whose output attributes are not contained in the list Z can be dropped from the mapper since the values that they produce will not be available for subsequent operations. Thus,

$$\pi_Z(\mu_F(s)) = \pi_Z(\mu_{F'}(s)) \qquad (2)$$

where $F' = \{f_{A_i} \in F \mid A_i \text{ contains at least one attribute in } Z\}$.

Selections When applying a selection to a mapper, we can take advantage of the fact that many attributes are mapped by arithmetic expressions. Some are even simple identity functions. A selection $\sigma_{C_{A_i}}$ where C_{A_i} is a condition on the attributes produced by some mapper function $f_{A_i} \in F$, can be pushed through a mapper. Hence,

$$\sigma_{C_{A_i}}(\mu_F(s)) = \mu_F(\sigma_{C'_{A_i}}(s)) \qquad (3)$$

where C'_{A_i} is a rewritten condition that uses the attributes of the input relation schema.

Joins It is often the case that mappers are applied to joins resulting in the expression $\mu_F(r \bowtie s)$. Depending of the type of join being performed the output of the relation $r \bowtie s$ can be very large. In these cases, whenever the join is being performed in attributes mapped by identity mapper functions, it is possible to use the rule

$$\mu_F(r \bowtie s) = \mu_F(r) \bowtie \mu_F(s) \qquad (4)$$

where the mapper functions in F do not produces duplicate values.

3.1 Plan selection

The choice of a particular plans is governed by the minimization of a metric of cost. The cost of a mapper operator depends fundamentally on the costs of evaluation each mapper function and on the cardinality of the input relation. For more details about the cost model for the mapper operator, please refer to (Carreira et al., 2006).

In order to estimate the cost of operators whose input is produced by mappers, the cardinality of the output relation produced by a mapper also needs to be estimated. This estimation, is an interesting problem in itself, because mappers can generate variably multiple output tuples for each input tuple. One way to approach this issue consists of estimating the average *mapper fanout* factor[1]. If a mapper is being executed for the first time, an initial estimate for its fanout needs to be computed. This can be done by combining the estimated fanout factors of the mapper functions involved in the mapper operator. Another interesting observation is that when mapper functions return empty sets, no output tuples are produced. Thus, the mapper in some situations may act as a filter, which turns the *selectivity* of the mapper into another relevant factor. Like fanout, the initial mapper selectivity can also be estimated from the selectivities of the mapper functions.

4 EXPERIMENTS

In this section we analyze the performance of the mapper operator and consider the gains obtained with the proposed logical optimizations. Our results indicate that one-to-many data transformations can be evaluated substantially faster than traditional database solutions like table functions or recursive queries. Moreover, we shall see that the optimizations defined for mappers impart performance gains that are not matched by traditional RDBMS solutions.

To that aim, we contrast alternative implementations of the data transformation proposed in Example 1.1 using the mapper operator with alternative implementations developed as table functions and recursive queries using two leading commercial RDBMSs. For more details on how to implement one-to-many transformations using RDBMSs, Please refer to (Carreira et al., 2005a). The mapper operator was implemented top of the XXL DBMS library (van den Bercken et al., 2001) which provides database query processing and optimization functionalities.

Figure 4: Average throughput of the different implementations of Example 1.1 tested with input relation sizes varying from 100K to 5M rows graphed with standard deviation. No query results are reported for recursive queries on the OEX system since the subset of recursive queries supported by OEX is not powerful enough for expressing one-to-many transformations.

The database implementations were tested on two systems henceforth designated as DBX and OEX[2]. The parameters of both RDBMSs were carefully aligned and the same I/O conditions were enforced through the usage of the same raw devices. The hardware used was a single CPU machine (running at 3.4 GHz), with 1GB main memory RAM, and Linux (kernel version 2.4.2) installed. Concerning workload, a synthetic version the input relation `LOANS` used in Example 1.1 was employed. To equalize the record length on XXL, DBX and OEX, a dummy column was added to the input table.

4.1 Results

We compare *throughput*, i.e., the amount of work done per second, of the distinct implementations of one-to-many data transformations. Throughput is expressed as the ratio of source records transformed per second and it is computed by measuring the *response time* of data transformation that consists of reading the input table, transforming it and materializing the output table. All the timings reported were obtained with logging disabled.

In the first experiment, we intended to test the raw performance of the mapper operator for the three distinct implementations. The results depicted on Figure 4 shows that one-to-many data transformations implemented with the mapper operator is more than 2 times better than table functions over DBX, which is the best alternative using RDBMSs. Since the amount I/O incurred by all the systems is similar, even considering the overhead of the RDBMSs by comparison with XXL, we conjecture that one-to-many data transformations implemented as mappers running inside the RDBMS are very efficient. We also considered the implementations using stored procedures. However, it turns out that the perfor-

[1] Similarly to (Chaudhuri and Shim, 1993), we designate the average cardinality of the output produced for each input tuple by mappers and mapper functions as *fanout*.

[2] Due to the restrictions imposed by the license agreements, the true names of the systems under test cannot be revealed.

Figure 5: Evolution of response time of applying selections to one-to-many transformations for increasing selectivity factors plotted in logarithmic scale. The mapper's original expression refers to $\sigma_{\texttt{ACCTNO}>p}(\mu_{acct,amt}(s))$, the mapper optimized expression refers to $\mu_{acct,amt}(\sigma_{\texttt{ACCT}>p}(s))$, the table function original expression refers to $\sigma_{\texttt{ACCTNO}>p}(TFacct, amt(s))$ and the table function optimized expression refers to $TFacct, amt(\sigma_{\texttt{ACCT}>p}(s))$. The size of the input relation s is fixed to 1M tuples.

mance is quite poor because logging cannot be disabled during their execution.

In the second experiment, we analyzed the potential gains of logical optimizations like those suggested in Section 3. To that aim, we considered the evaluation of the expression $\sigma_{\texttt{ACCTNO}>p}(\mu_{acct,amt}(s))$ together with its optimized equivalent $\mu_{acct,amt}(\sigma_{\texttt{ACCT}>p}(s))$ obtained by pushing down the selection. The constant p was used to induce different selectivities. Moreover, we consider that the function *acct* performs a direct mapping, i.e., is an identity function. In Figure 5, we depict the performance of both the original and the optimized expressions with varying selectivities. We observe that smaller selectivities correspond to the highest gains of the optimized expression over the original. For comparison, we draw the evolution of the selection applied to one-to-many transformations implemented using table functions on the OEX system, represented as $\sigma_{\texttt{ACCTNO}>p}(TFacct, amt(s))$. The performance of this solution is similar to the unoptimized version of mapper and only for small selectivities. This is due to the fact that the whole relation is read but few output tuples are generated.

Surprisingly, by analyzing the query plans generated by RDBMSs, we came across the fact that, whenever table functions or recursive queries are used to encode one-to-many data transformations, neither DBX nor OEX are capable of pushing down a selection on a directly mapped attribute. Hence, for comparison, we also tested the corresponding optimized expression $TFacct, amt(\sigma_{\texttt{ACCT}>p}(s))$ obtained manually. We observed that the manually optimized expression of the table functions bring higher gains specially on relatively high selectivities. For high selectivities, the response time of the original (non-optimized) RDBMS solution increases sharply. We conjecture that this behavior has to do with idiosyncrasies of OEX related with the pipelining of the tuples resulting from the table function into the selection operator.

5 CONCLUSION

In this paper we focused on the feasibility of incorporating a specialized operator for handling one-to-many data transformations on RDBMSs. This extension is attractive, not only because one-to-many data transformations cannot be expressed using relational algebra but also because data usually resides in an RDBMS. We outlined the concrete syntax for this operator and then examined how a query optimizer can be extended to consider more advantageous execution plans in the presence of mappers. To test our ideas we analyzed experimentally different implementations of one-to-many data transformations using mappers and contrasted them with traditional implementations using table functions and recursive queries using two industry-leading RDBMSs. To the best of our knowledge, this is the first experimental assessment of one-to-many data transformations on RDBMSs.

The experiments showed that a native implementation of the mapper operator outperformed the best RDBMS solution by almost 3 times. We have also observed that RDBMSs do not in general perform even very simple but highly valuable optimizations when table functions and recursive queries are used. Thus, we posit that one-to-many data transformations expressed by combining standard relational operators and mappers constitute a valid alternative.

The simple iterator-based semantics of the mapper operator enables efficient executions of one-to-many data transformations and favors an easy integration into the query processor of a database system. Towards physical optimization, we are developing different algorithms for the mapper operator to take advantage of duplicate values by employing caching techniques and hybrid-hashing proposed by (Hellerstein, 1998). Additionally, we consider incorporating the mapper operator in Apache Derby open source RDBMS (Apache, 2005).

One limitation of our work is that, despite the effort to configure the different systems so that they run in similar conditions, the alignment of these configurations lacks quantification. To address this shortcoming we consider running TPC-H (TPC, 1999) loads on the different systems in order to obtain a metric for comparing their re-

spective configurations.

REFERENCES

Aho, A. V. and Ullman, J. D. (1979). Universality of data retrieval languages. In *Proc. of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Lang.*, pages 110–119. ACM Press.

Amer-Yahia, S. and Cluet, S. (2004). A declarative approach to optimize bulk loading into databases. *ACM Transactions of Database Systems*, 29(2):233–281.

Apache (2005). Derby homepage. http://db.apache.org/derby.

Carreira, P., Galhardas, H., Lopes, A., and Pereira, J. (2005a). Extending the relational algebra with the Mapper operator. DI/FCUL TR 05–2, Department of Informatics, University of Lisbon. URL `http://www.di.fc.ul.pt/tech-reports`.

Carreira, P., Galhardas, H., Lopes, A., and Pereira, J. (2006). One-to-many transformation through data mappers. *Data and Knowledge Engineering Journal (DKE)*, Elsevier Science.

Carreira, P., Galhardas, H., Pereira, J., and Lopes, A. (2005b). Data mapper: An operator for expressiong one-to-many data transformations. In *7th Int'l Conf. on Data Warehousing and Knowledge Discovery, DaWaK '05*, volume 3589 of *LNCS*. Springer-Verlag.

Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *PODS '98: Proc. of the ACM Symp. on Principles of Database Systems*, pages 34–43. ACM Press.

Chaudhuri, S. and Shim, K. (1993). Query optimization in the presence of foreign functions. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'93)*, pages 529–542.

Cui, Y. and Widom, J. (2001). Lineage tracing for general data warehouse transformations. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*.

Eisenberg, A., Melton, J., Michels, K. K. J.-E., and Zemke, F. (2004). SQL:2003 has been published. *ACM SIGMOD Record*, 33(1):119–126.

Galhardas, H., Florescu, D., Shasha, D., and Simon, E. (2000). Ajax: An extensible data cleaning tool. *ACM SIGMOD Int'l Conf. on Management of Data*, 2(29).

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2002). *Database Systems – The Complete Book*. Prentice-Hall.

Haas, L. M., Miller, R. J., Niswonger, B., Roth, M. T., Schwarz, P. M., and Wimmers, E. L. (1999). Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Engineering Bulletin*, 22(1):31–36.

Hellerstein, J. M. (1998). Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 22(2):113–157.

Lomet, D. and Rundensteiner, E. A., editors (1999). *Special Issue on Data Transformations*, volume 22. IEEE Data Engineering Bulletin.

Melton, J. and Simon, A. R. (2002). *SQL:1999 Understanding Relational Language Components*. Morgan Kaufmann Publishers, Inc.

Rahm, E. and Do, H.-H. (2000). Data Cleaning: Problems and current approaches. *IEEE Bulletin of the Technical Comittee on Data Engineering*, 24(4).

Raman, V. and Hellerstein, J. M. (2001). Potter's Wheel: An Interactive Data Cleaning System. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*.

TPC (1999). Benchmark H standard specification. http://www.tpc.org.

van den Bercken, J., Dittrich, J. P., Kräamer, J., Schäafer, T., Schneider, M., and Seeger, B. (2001). XXL A library approach to supporting efficient implementations of advanced database queries. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*.