Optimisation d'applications orientées données

Sonia Guéhis[1] Philippe Rigaux[1] Virginie Thion-Goasdoué[1]

September 3, 2009

Abstract

Nous nous intéressons à la classe des programmes appliqués aux bases de données (DP[1]). Un tel programme contient des appels (sous forme de requêtes SQL) à une base de données hébergée sur un serveur. Pour chaque requête, le coût d'exécution associé est composé du coût d'exécution de la requête en elle-même ainsi que du coût de soumission de la requête au serveur et de transfert des données, via le réseau, entre le programme et le serveur. Par défaut, un DP génère un grand nombre de requêtes SQL élémentaires vers le serveur. Cette méthode prive le serveur de la possibilité d'optimiser les requêtes de façon globale. Nous proposons ici une méthode permettant d'optimiser *globalement* un programme, sans en modifier le code, par minimisation du coût des échanges entre le programme et le serveur. À partir du *résumé* du programme, nous identifions différentes stratégies de matérialisation des données. La définition d'un modèle de coût associé à chaque stratégie nous permet ensuite de choisir une stratégie globale avantageuse. Des expérimentations montrent que la méthode que nous proposons peut améliorer le temps d'exécution d'un DP jusqu'à plusieurs ordres de grandeur.

Keywords: database applications, data retrieval optimization, object-relational mapping.

1 Introduction

Relational database systems dispose of a sophisticated optimization engine, capable of efficiently handling intricate queries that access many tables and express complex conditional statements. However, most often, SQL queries are not executed independently but embedded in traditional programming languages such as Java, C++ or PHP. Embedded SQL programs, called *database programs* in the present paper, tend to use specific navigation patterns. Typically, the code organization is driven by the logic of the application, and not by the expected efficiency of the queries. As a result, the program execution consists of a flow of small queries successively submitted to the data server, and poorly exploits its optimization power. This calls for a *global* optimization of the program that attempts to minimize both the number of submitted queries, and the number of fetch requests.

[1] LAMSADE, Université Paris-Dauphine, Place du Maréchal de Lattre de Tassigny 75775 Paris Cedex 16, France. <firstname.lastname>@dauphine.fr

[1] DP pour *"database program"*.

Such an optimization cannot be the concern of the application programmer, though. Indeed, writing code that minimizes the number of queries requires a sophisticated programming approach. It leads to execute a few, large queries that retrieve, in one shot, large chunks of the data required by an application. These chunks must then be organized in a convenient data structure which is finally traversed and processed. This constitutes a non-natural and difficult way of building programs.

We need a mechanism that combines the conceptual simplicity of an individual data access based on object-like navigation, convenient to the programmer, with a set-based retrieval strategy that limits the client-server exchanges and promotes server-side optimization. The first issue is already addressed by several persistency layers providers (Hibernate, Ruby-on-the-rail, The Zend Framework) which rely on the so-called Object-Relational Mapping (ORM) approach. However, the set-based evaluation of queries is only partially covered in these frameworks, with ad-hoc and local solutions.

In the present paper we propose an optimization approach for database programs based on a declarative description called *program summary*. It describes the sequence of accesses to the database instance, viewed as a graph supporting navigation primitives. This declarative specification lets the optimizer manipulate data retrieval requirements as a whole, instead of processing them as a flow of SQL queries submitted independently to the database system. It becomes then possible to choose a global strategy that minimizes the cost of transferring data from the server to the client. A key feature is that the summary can be created and modified independently from the program logic. This saves programmers from worrying about optimization issues, and lets experts adapt the execution strategy at any moment of the application lifetime.

We analyse the characteristics of embedded query execution, and identify the main time-consuming operations that contribute to the global execution cost. We derive from this analysis a cost model to assess the run-time behavior of a specific execution. Based on this cost model, we develop a rewriting strategy to obtain an optimized scheduling of embedded queries execution for a given input database program.

Finally, we conduct a set of experiments on the TPC-H Benchmark, applying our optimization approach to Java/JDBC applications. Our implementation relies on execution primitives provided by the Hibernate ORM layer. First, this confirms that our optimization strategy can be implemented with standard engineering tools. Second, our results show impressive improvements (one or two orders of magnitude faster than the standard approach). This optimization is obtained at a very low cost: it requires the specification of the summary and the initial execution of an optimization module that tunes the Hibernate execution strategy.

In the rest of the paper we first shortly develop the problem statement (Section 2). We then successively present the components of our proposal. Section 3 is devoted to our model and formalizes program summaries. Section 4 proposes our cost model and the associated evaluation strategy. We give our experimental setting and results in Section 5. Finally, Section 6 discusses related work and Section 7 concludes the paper.

Customer (custKey, address, nationKey, phone, acctbal, mktsegment, comment)
Orders (orderKey, custKey, orderStatus, totalPrice, orderDate, orderPriority, clerk, shipPriority, comment)
LineItem (orderKey, linenumber, partKey, suppKey, quantity, extendedPrice, discount, tax, returnFlag, lineStatus, .., comment)
Part (partKey, name, mfgr, brand, type, size, container, comment, retailPrice)
Nation (nationKey, name, regionKey, comment)
Region (regionKey, name, comment)

Figure 1: Schema of the TPC-H benchmark

2 Problem Statement

This section provides a short review of the technical aspects of SQL programming, including an introduction to the two main techniques in use nowadays: cursor-based navigation, and Objet-Relational Mapping (ORM) interfaces. Throughout the paper, we use as a running example the schema of the TPC-H Benchmark [19], given in Figure 1.

2.1 Cursor-based navigation: the $n + 1$ queries problem

Consider a program that produces invoices to customers. Its typical structure is summarized by the pseudo-code shown in Figure 2. In terms of data accesses, the program can be described with SQL queries and the three well-known cursor primitives **open**, **fetch** and **close**. The primitives are processed by a driver component incorporated in the client application. The standard behavior of SQL drivers is to send a remote instruction to the data server, for each call to either **open** or **fetch**. This involves repeated query compilations and network round-trips. Incidental evidence shows that they constitute a major factor in the performance of SQL programs execution, much more important than the server-side execution of the query itself. [5] reports for instance a 99% overhead of network communication over query time. See also [18], page 165, on the cost of cursor iterations.

In technical terms, the situation is most often referred to as the "n + 1 query problem", denoting the execution of a nested loop join on the client side. Assume for instance that the program processes 1,000 invoices. There is one execution of query q_1 for retrieving the customers, and 1,000 executions of queries q_2 and q_3 for, respectively, the nations and orders of each customer. Each execution of either q_2 or q_3 retrieves only a very small part of the total information needed by the program. The full set of nations or orders could actually be easily obtained with a single SQL query. Splitting the data retrieval in many small pieces constitutes a major penalty. As an illustration, one of our experiments shows that the direct execution of our invoice program takes 28 mns (!) on a relatively small MySQL database with 20,000 customers. Loading all the required data in the application program with a single large query takes only 35 s. Similar results are observed for other database servers.

Proliferation of small queries could be avoided and replaced by the execution of a few, large queries. Unfortunately, this contradicts the natural and safe programming practice of retrieving an information only when it needs to be processed. The programmer should not worry about data management issues (e.g., storing data in

```
open ($q_1$: select * from Customer where mktsegment='tourism')
while (C := fetch($q_1$)) do
   // Do something with C
    [...]
   // Access to the nation
   open ($q_2$: select * from Nation where nationKey=C.
       nationKey)
   N := fetch($q_2$)
   close ($q_2$)
   // Loop on C's orders
   open ($q_3$: select * from Order where custKey=C.custKey)
   while (O := fetch ($q_3$)) do
      // Access to the line items
     open ($q_4$: select * from LineItem where orderKey=O.
         orderKey)
     while (L := fetch($q_4$)) do
      // access parts, etc.
     enddo
    close ($q_4$)
   enddo // End loop on orders
   close ($q_3$)
enddo // End loop on customers close($q_1$)
```

Figure 2: A simple invoice program.

temporary structures for future use), but rather focus on the mere application logic.

Our sample program can be equivalently rewritten (in terms of data retrieval) with only one query, using joins and outer joins (Figure 3). Although this approach does avoid the n+1 query pitfall, an immediate downside is that each row *T* obtained in the **fetch** loop mixes information on customers, nations, orders and line items. Moreover, each customer is repeated with each of his *Order* occurrences, and both are repeated for each order's item. Manipulating this representation involves an intricate and redundant decoding process which is incompatible with basic software design principles.

2.2 Interfacing with Objet-Relational Mappings

Programming techniques that convert data between relational databases and object-oriented programming languages are collectively known as Object-Relational Mapping (ORM in short). Basically, the idea is to create a virtual object database which is materialized on demand. Most of the software development platforms (e.g., J2EE/Hibernate [11], PHP/Zend [15], Ruby and Rails [21], etc.) come equipped with an ORM layer that helps to better integrate application logic with data access and manipulation.

Figure 4 shows the invoice program written in ORM style. A clear advantage is that SQL queries are now completely hidden under an object-based navigation process. However, with respect to the underlying query execution scheduling, the difference remains mostly syntactical. The ORM approach clearly favours a scattered execution of trivial queries. In the worse case, each navigation operation towards an object (e.g., from a customer to its nation) or a collection (e.g., from a customer to his orders) is implemented with a dedicated SQL query, which results in an execution process essentially similar to the cursor-based version of our

```
open ($q_1$: select * from Customer as C join Nation on
nationKey=C.nationKey
outer join Order as O on O.custKey=C.custKey
outer join Lineitem as L on L.orderKey=O.orderKey
    where mktsegment='tourism')
while (T := fetch($q_1$)) do
   // T contains customer + nation + order + line item
   [Decode T to obtain the customer C -- Process C (when
       met for the first time)]
   [Decode T to obtain the nation N -- Process N]
   [Decode T to obtain the order O -- Process O]
   [Decode T to obtain the line item L -- Process L]
   [...]
enddo // End loop on customers close($q_1$)
```

Figure 3: The invoice program, rewritten with outer joins

```
for each (C in CustomerCollection.filter(mktsegment='
    tourism')) do
   // Do something with C
   [...]
   // Access to the nation
   N := C.nation;
   // Loop on C's orders
   for each (O in C.orders) do
       // Process orders
   enddo // End loop on orders
enddo
```

Figure 4: The invoice program implemented over an ORM layer.

program, or even worse (e.g., one query per record when records are made into objects).

The ORM paradigm constitutes nevertheless a first step towards an asynchronous execution of data manipulations and data server requests. Advanced ORM layers, such as Hibernate, provide some preliminary support for either prefetching data or storing in a client cache a part of the database which is repeatedly accessed by a program. Choosing an appropriate strategy remains however a matter of expertise, and depends on hints expressed at the schema level, and not at the program level. Besides, it is unclear how the relevant choices can be generalized so as to cover a wide and generic class of programs. Our approach, outlined below, aims at filling this gap.

2.3 Approach overview

The architecture is summarized in Figure 5. Client applications consist of two layers . The upper one deals with the tasks related the *application logic*. From this layer point of view, accessing the database is reduced to navigating in a local data graph managed by the lower layer called *object layer*. It stores a local cache containing a subset of the database instance used by the application. This cache is structured as a graph, called *program trace* in the following, where nodes represent objects, and edges represent relationships. The object layer is in charge of interacting with the data server and provides navigation primitives on the program trace. It acts therefore as an ORM layer.

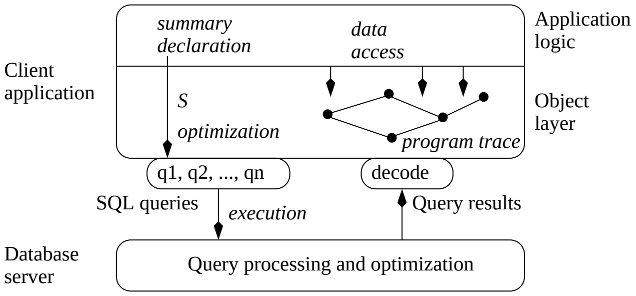

Figure 5: System architecture

The main issue is to decide when and how the program trace is materialized. A simple solution is a lazy strategy (the default approach of ORM systems) which executes a query on a collection when an element of the collection is not found in the local cache. This gives rise to the n+1 query problem explained above.

In order to promote a global strategy based on a full view of the program behavior, we define the content of the program trace by a high-level declaration called *summary*. The summary is transmitted by the upper layer to the object layer component at the beginning of an execution, and is used by the object layer as a global specification that serves to determine an appropriate materialization strategy for the program trace. Basically, the object layer carries out an optimization mechanism that produces a set $q_1, q_2, \ldots, q_n$ of SQL queries along with scheduling rules. Each query is in charge of producing a specific part of the program trace, and must be executed when this part needs to be materialized.

An important factor to consider is the cost of maintaining the local cache. The **decode** operation shown in Figure 5 must be processed when a query result is transmitted to the client. This query may be a join, which creates rows in the result set that mix several objects from the data graph. Consider for instance a join between *Customer*, *Nation* and *Order*. Decoding a row from the result set involves an extraction of a single node for the customer, a node for the nation, and one node for each order. Edges between these nodes must be created as well. Although this decoding can be automatically handled by the object layer, its processing overhead must be considered.

3 Program summaries

3.1 The data graph

We model the database as a directed graph and each tuple in the database as a vertex in the graph. Each link (foreign-key, primary-key) is modeled as a directed edge between the corresponding tuples. The graph is virtual, and must be partially materialized during the program evaluation process.

A part of the data graph is shown in Figure 6 for a TPC-H database. The main feature of this representation is the simplicity and concision of the concepts in

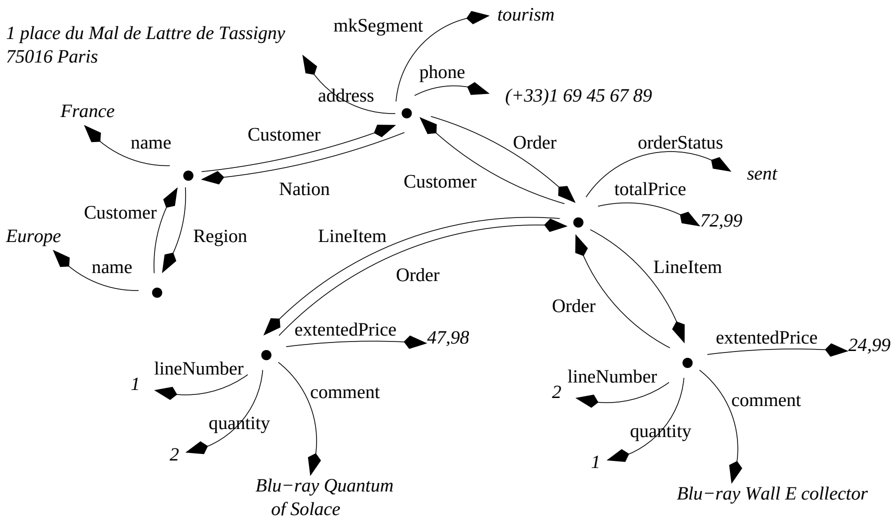

Figure 6: A part of the virtual graph

use: each vertex corresponds either to a tuple or a value; each edge is labeled and represents an association between vertices. The edges cover all the various links which are usually distinguished in a relational database, namely tuple-to-attribute and tuple-to-tuple.

In this spirit, we define an associated notion of schema. Let $\mathcal{T}, \mathcal{R}, \mathcal{A}$ be sets of symbols pairwise disjoint, $\mathcal{T}$ finite, and $\mathcal{R}, \mathcal{A}$ countably infinite. The elements of $\mathcal{T}$ are *atomic types*, those in $\mathcal{R}$ *relation names*, and those in $\mathcal{A}$ *attribute names*. A *(graph database) schema* is a directed labeled graph (V, E, λ, μ) where $V \subseteq \mathcal{T} \cup \mathcal{R}$ is a set of vertex, and $E \subseteq (V \cap \mathcal{R}) \times V$ is a set of edges. λ is a labeling function from E to relation or attribute names, and μ is a multiplicity function from E to $\{1, *\}$. If $\mu(e) = 1$, this indicates that there can be at most one instance of e in the database for a given initial vertex; if $\mu(e) = *$, multiple instances are allowed. Figure 7 shows a part of the graph schema of our sample database.

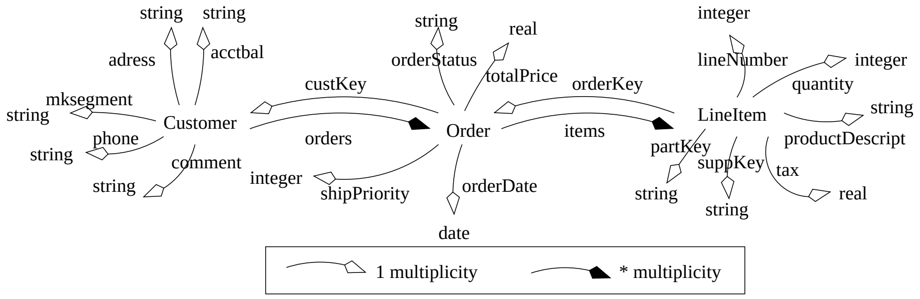

Figure 7: The schema of the data graph

Given a relational database, there exists a straightforward mapping between the relational schemas and instances and the graph schemas and instances.

3.2 Program summaries

A *program summary* is a concise and simple representation of the part of the database instance accessed by a program P during its execution. A summary S is built from *path expressions*. Syntactically, paths expressions correspond to a subset of the XPath language [20]. In its simplest form, a path expression is a sequence of labels of edges pairwise connected in a graph schema. A path expression may contain *predicates* which are Boolean combinations of atomic formulas of the form $q = value$ where q is a path expression. The general form of a path expression is $l_1[p_1].l_2[p_2].\cdots.l_n[p_n]$ where each p_i such that $1 \leq i \leq n$ is a predicate and each l_i, $1 \leq i \leq n$ is either a label, either the root vertex symbol db.

Example 1 *Here are some examples of path expressions over our sample schema.*
$db.Order[OrderStatus = 'sent'].LineItem.quantity;$
$db.Order[OrderStatus = 'sent'];$
$Customer[Nation.Region.name = 'Europe'].Order.LineItem.$

A program summary is simply a tree of path expressions.

Definition 1 (Summary) *A* summary *is inductively defined as follows :*

- *if expr is a path expression, then* $@expr\{\}$ *is a summary;*
- *if* p *is a path expression and* $\{s_1, \cdots, s_k\}$, $k \geq 0$ *are summaries, then* $@p\{s_1; \cdots; s_k\}$ *is a summary.*

The semantics of the language corresponds to nested loops that explore the data graph, one loop per path expression. This navigation produces the *trace* of a summary S, which is a finite unfolding of the graph $\mathcal{G}_I$ representing the nodes visited during the evaluation of S. Formally:

Definition 2 (Trace of a summary) *Let* S *be a summary, represented as a tree of path expressions. Let* $\mathcal{G}_I$ *be a data graph and* v *be one of its vertices. The* trace $\mathcal{T}_{S,v}(\mathcal{G}_I)$ *of* S *on node* v *with respect to* $\mathcal{G}_I$ *is a tree, subgraph of* $\mathcal{G}_I$ *and rooted at* v*, defined inductively as follows:*

1. *if* S *is* `@l{}` *where* l *is a label, then* $\mathcal{T}_{S,v}(\mathcal{G}_I)$ *is the subgraph of* $\mathcal{G}_I$ *containing all the edges* $v \xrightarrow{l} v'$*;*
2. *if* S *is* `@p.l{}` *where* p *is a path expression and* l *is a label, then* $\mathcal{T}_{S,v}(\mathcal{G}_I)$ *is* $\mathcal{T}_{@p\{\},v}(\mathcal{G}_I) \cup \{v' \xrightarrow{l} v'', v' \in terminal(\mathcal{T}_{@p\{\},v}(\mathcal{G}_I))\}$
3. *if* S *is a summary of the form* $@p\{s_1; ...; s_k\}$ *then* $\mathcal{T}_{S,v}(\mathcal{G}_I) = \biguplus_{1 \leq i \leq k} \mathcal{T}_{@p.s_i\{\},v}(\mathcal{G}_I)$

The trace $\mathcal{T}_S(\mathcal{G}_I)$ *of* S *is defined by* $\mathcal{T}_{S,\texttt{root}}(\mathcal{G}_I)$ *(where* `root` *is the pseudo-root of* $\mathcal{G}_I$*).*

Our approach associates a summary S to a program P. The trace of S defines a subset of the data graph that needs to be transferred from the server to the client during the execution of P. Figure 8 shows a summary of the invoice program and its interpretation.

```
@db.Customer[mktsegment='tourism']{
  @Nation{};
  @Order{
    @LineItem[lineNumber=2]{
     @Part{}
    }
  }
}
```

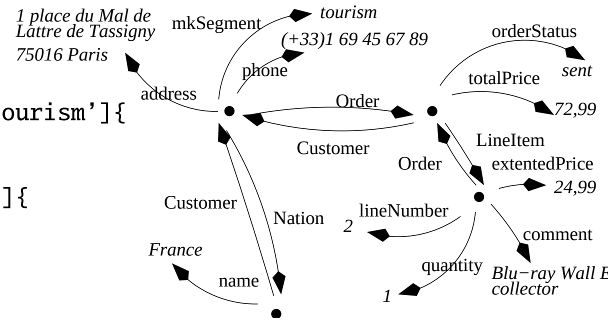

Figure 8: A program summary and its interpretation

Note that we do not require a summary to be *complete*, i.e., *P* may need to access some parts of the database instance which are beyond the trace of its summary. This is not a problem, since the default client/server interaction (i.e., lazy execution of SQL queries) can be used in such cases. However, when *P* requires some data in the semantics of its declared summary, this information should be found in the local cache. The summary must therefore cover to the costly part of the program execution.

4 Materialization

In the present section, we analyse the optimization space, propose a cost model and define an optimization strategy. All the examples are based on the invoice program.

4.1 The optimization space: outer join, union and decorrelation

Rewriting rules take a program summary S and generate an SQL execution which materializes the trace of S. The materialization process can be described in terms of SQL queries, **open** and **fetch** operations, and finally a **decode**($\mathcal{T}_P$, r, t) operator which takes as input a row r from a result set, extracts from r a node of type t, and inserts this node in the trace $\mathcal{T}_P$.

The basic rewriting strategy, denoted LAZY in the following, retrieves the data node only when they are needed. As discussed in Section 2, it generates a large number of query executions and client-server round-trips.

Rule 1 (Lazy) *The trace of* `@t1[p1]{@t2[p2]}` *can be materialized by the following SQL program:*

```
open ($q_1$: select * from t1 where p1) while ($r_1$ :=
fetch($q_1$)) do
    decode ($\mathcal{T}_P$, $r_1$, $t_1$)
    open ($q_2$: select * from t2 where p2 and t2.id_t1 = $r_1$.id)
    while ($r_2$ := fetch($q_2$)) do
      decode ($\mathcal{T}_P$, $r_2$, $t_2$)
    enddo
enddo
```

Rule 1 generalizes to summaries of the form `@t1[p1]{@t2[p2]; ···; @tn[pn]}` by adding inner loops. It corresponds to the "natural" way of writing a database program, fetching a row just in time, and throwing it out when it has been exploited. The next rule materializes the trace of a summary with a single query based on outer joins.

Rule 2 (Outer joins) *The trace of* `@t1[p1]{@t2[p2]}` *can be materialized by the following SQL program:*

```
open ($q_1$: select * from t1 outer join t2 on t1.id=t2.id_t1
    where
p1 and p2) while (r := fetch($q_1$)) do
    decode ($\mathcal{T}_P$, r, $t_1$)
    decode ($\mathcal{T}_P$, r, $t_2$)
enddo
```

Rule 2 generalizes to summaries of the form `@t1[p1]{@t2[p2]; ···; @tn[pn]}` by adding the **Union** operator. The (unique) SQL query becomes:

```
 select (expression) from t1 outer join t2 on t1.id=t2.id_t1
    where p1 and p2
  union
   ...
  union
 select (expression) from t1 outer join tn on t1.id=tn.id_t1
    where p1 and pn
```

where *expression* standardizes schemas of tables `t1`...`tn`, by adding `null` values on unmatchable attributes.

The code shown in Figure 3 retrieves the data of the invoice program thanks to his rewriting strategy, denoted Single. A clear advantage is the minimization of the number of query executions which reaches here his minimal value, 1. The number of fetches is also reduced with respect to the lazy evaluation. The overhead with this rule is the cost of the **decode** operation which must deal with large rows containing several nodes. Two distinct rows may also contain the same node (e.g, a customer repeated with each of his orders), leading to redundant decoding.

Finally, the last rule that we consider is the *decorrelation rule* (Decorr). It evaluates separately one query for each database table involved in the production of the program trace.

Rule 3 (Decorrelation) *Let* `@t1[p1]{@t2[p2]}` *be a summary. The trace can be materialized by the following SQL program:*

```
open ($q_2$: select * from t2 where p2 and id in (select * from
    t1
where p1)) while ($r_2$ := fetch($q_2$)) do
     decode ($\mathcal{T}_P$, $r_2$, $t_2$)
enddo open ($q_1$: select * from t1 where p1) while ($r_1$ :=
fetch($q_1$)) do
     decode ($\mathcal{T}_P$, $r_1$, $t_1$)
enddo
```

Consider for instance the summary `@Customer[mktsegment='tourism'] {@Nation}`. The materialization process based on Rule 3 executes the following queries:

```
q1: select * from Customer where mktsegment='tourism'
q2: select * from Nation where nationKey in (select
    nationKey from Customer where mktsegment='tourism')
```

Rule 3 generalizes trivially to summaries of the form `@t1[p1] {@t2[p2]; ···; @tn[pn]}`. Although this seems to yield complex queries, this technique limits the cost of **decode** because each **fetch** retrieves a row which directly corresponds to a unique node in $\mathcal{T}_P$.

4.2 The cost model

We now analyse the cost of materialization strategies that result from the application of rewriting rules. We assume that (i) the cost of executing a query on the server side is constant and does not depend on the query size, and (ii) the cost of transmitting a row from the server to the client is independent from the row size. Part of our experiments will be devoted to confirming these assumptions. We model the cost of the query evaluation process as a linear combination of three key operations: **open**, **fetch**, and **decode**:

$$cost = C_{open} \times N_{open} + C_{net} \times N_{fetch} + C_{decode} \times N_{decode} \qquad (1)$$

C_{open}, C_{net}, and C_{decode} represent the relative importance of each key operation. They depend on many factors, including the programming language, the network bandwidth, the database server efficiency, etc. We postpone their study to the performance evaluation, and focus on an estimation of N_{open}, N_{fetch} and N_{decode} for a basic program summary `@t1[p1]{@t2[p2]}`.

In the following, $|t_1|$ (resp. $|t_2|$) denotes the number of rows of t_1 (resp. t_2) in the trace of the summary, and α the average number of rows from t_2 matching a row from t_1 (i.e., the average number of executions of the inner loop in Lazy mode). Table 1 summarizes the values of N_{open}, N_{fetch} and N_{decode} for Lazy (Rule 1), Single (Rule 2) and Decorr (Rule 3).

The Lazy mode exhibits the larger number of **open**, because a query on t_2 is executed for each row of t_1. The number of **fetch** is $|t_1|$ * (1 + Max(α, 1)). t_1 is fetched once in the outer loop, and there is at least one **fetch** in the inner loop, even if $\alpha = 0$ (i.e., no matching row in t_2). In such a case, the first **fetch** executed in the inner loop returns a `null` object, yet it still counts for one exchange with the data server.

Single yields exactly one **open**, and thus completely avoids the N+1 query problem. N_{fetch} is equal to Max($|t_1|$, $\alpha|t_1|$) because of the outer join. The materialization process is identical to the Lazy mode. Finally, the Decorr strategy retrieves separately t_1 and the related rows in t_2, for a total number of fetches equal to $|t_1| + |t_2|$. Each row is fetched once, and decoded independently.

Both Single and Decorr avoid the N+1 queries problem. The difference lies in the trade-off between the number of **fetch**es, which is always lower for Single, and the cost of **decode**, which depends on α. We distinguish two cases. If $\alpha \in [0, 1]$, there is at most one row from t_2 matching a row from t_1. In that case the cost of

	N_{open}	N_{fetch}	N_{decode}
LAZY	$1 + \|t_1\|$	$\|t_1\|\,(1 + \mathrm{Max}(1, \alpha))$	$\|t_1\|\,(1 + \alpha)$
SINGLE	1	$\mathrm{Max}(\|t_1\|, \alpha\|t_1\|)$	$\|t_1\|\,(1 + \alpha)$
DECORR	2	$\|t_1\| + \|t_2\|$	$\|t_1\| + \|t_2\|$

Table 1: Cost analysis of the rewriting strategies for the summary `@t1[p1]{@t2[p2]}`

decode is $2 * |t_1|$ for SINGLE, and $|t_1| + |t_2| < 2 * |t_1|$ for DECORR. The latter choice is likely to be the better.

Example 2 *Consider the summary* `@customer{@nation}` *over the TPC-H benchmark. Since there is a many-to-one relationship between Customer and Nation,* $\alpha = 1$. *Now, assume an instance with* $10,000$ *customers and* 200 *nations.* SINGLE *performs an outer join, with* $10,000$ ***fetch*** *and* $20,000$ ***decode****. Note that a same nation is decoded, on average, 50 times.*

DECORR *executes 2 queries, and at most* $10,200$ ***fetch*** *and* ***decode****. The number of* ***fetch*** *is slightly larger with respect to* SINGLE *because customers and nations are retrieved separately, but the strategy avoids redundant decoding of nations.*

If $\alpha > 1$, there are, on average, several rows from t_2 matching a row from t_1. The cost of **decode** is $|t_1| * (1 + \alpha)$ both for SINGLE and DECORR. Since the former generates less **fetch**, it should be preferred.

Example 3 *Consider the summary* `@customer{@order}` *over the TPC-H benchmark. Since there is a one-to-many relationship between customer and order,* $\alpha > 1$. *Now, assume an instance with* $10,000$ *customers and* $50,000$ *orders where* $\alpha = 5$. SINGLE *performs an outer join, with* $50,000$ ***fetch*** *and* $60,000$ ***decode****.* DECORR *executes 2 queries,* $60,000$ ***fetch*** *and* ***decode****. The first strategy saves 10,000* ***fetch*** *and must be preferred.*

Based on to the previous analysis, the following guidelines can be adopted:

1. a node fetched through a many-to-one relationship must be kept in the cache until its parent loop is not completed: this is the case for instance for *Nation* nodes;
2. a node fetched through a one-to-many relationship needs not be kept in the cache once it has been processed by the program: this is the case for instance for the *Order* nodes.

4.3 The optimization algorithm

The algorithm takes as input a program summary S viewed as a tree of path expressions (the generalization to forests is immediate). It proceeds in two steps. First S is partitioned in *blocks*, each block being associated to a specific rewriting rule. Second the blocks are evaluated using a bottom-up order.

Figure 9 illustrates the technique applied to the invoice program. The summary S is shown on the left part. Each node is a table name in the TPC-H database. S is first labelled with relationship cardinalities which can be either * (one to many) or 1 (many to one). The partitioning procedure starts from the root node, *Customer*, scans the summary top-down, and ”cuts” the edges corresponding to a many-to-one relationship. One obtains a main block, `D`, and two sub-blocks, `B` and `C`, respectively rooted at *Nation*, and *Part*. The procedure is recursively called for each sub-block. This further decomposes the block `B` by creating a new sub-block rooted at *Region*.

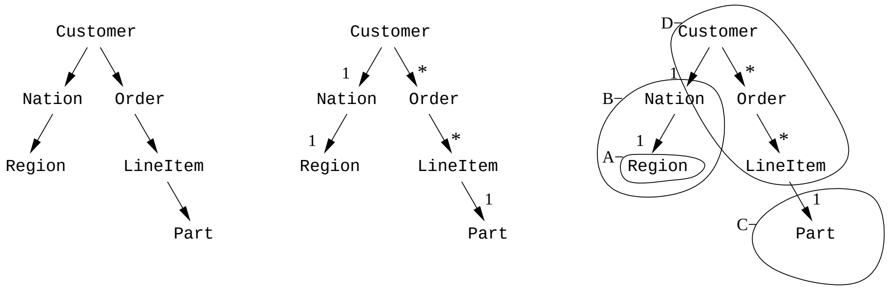

Figure 9: Illustration of the optimization algorithm

Each block contitutes a sub-summary which is materialized as follows:

1. Inside each block, Rule 2 (Single) is applied. Block `D` for instance yields an outer join *Customer-Order-LineItem*.

2. When a block b_2 is a child of a block b_1 (i.e., the summary is of the form `@b1{@b2}`), Rule 3 (Decorr) is applied. For instance a decorrelated query on *Part* is executed to materialize the trace of block `C`.

The scheduling of the materialization follows a bottom-up approach. Block `A` and `C` must be executed first. Their results are put in the cache. Block `B` follows, then block `D`. Note that when the loop on the outer join *Customer-Order-LineItem* is carried out, the related *Nation* and *Part* tuples can be found in the cache.

The optimization algorithm minimizes the number of decode operation (in fact, it is easy to see that each row in the trace is only decoded once). More generally, it satisfies the following property.

Proposition 1 (Global optimization of materialization) *Let I be an instance such that the average number of instances of any one-to-many relationship is greater than 1. Then, for any program summary S, the optimization algorithm minimizes* $N_{open} + N_{fetch} + N_{decode}$.

Assume first that the summary consists of a single block of the form `@t1[p1]{@t2[p2]}`. By definition of a block, the relationship is one-to-many and $\alpha > 1$. Referring to Table 1, it is easy to prove that the Single strategy minimizes $N_{open} + N_{fetch} + N_{decode}$. By induction on the form of the block, this extends to a block of any size.

Next, consider a summary with several blocks. Blocks can now be seen as tables and relations between blocks are many-to-one. Referring to Table 1, it is easy to prove that the Decorr strategy minimizes $N_{open} + N_{fetch} + N_{decode}$ for a simple summary of the form `@t1[p1]{@t2[p2]}` with $\alpha \leq 1$. By induction on the form of the summary, this extends to any summary.

The condition states that the instance is not degenerated, i.e., that it matches the schema structure in terms of cardinalities. The algorithm delivers then the best plan if we make the simplified assumption that the costs of **open**, **fetch** and **decode** are roughly similar. As mentioned above, these costs actually depend on many factors, but our evaluation strategy, designed to limit the impact of each factor, is expected to remain robust enough.

5 Experiments

We conducted extensive experiments to assess the gain obtained by our method. We first briefly describe our experimental setting, then discuss our results.

5.1 Experimental setting

The optimization approach has been implemented, in Java, as an extension of the Hibernate 3.3.0 ORM layer. Application programs can be written with standard Hibernate techniques, the only specificity being a call to the optimization module, at the beginning of the program, providing the summary. The module analyses the summary, applies the optimization algorithm, and performs the necessary actions to enable the resulting strategy. The main actions at this step consist either in executing some queries and putting their result in the cache or tuning some Hibernate parameters to obtain the required execution schedule.

The program and the data server run on two distinct computers, connected by a LAN (Local Area Network) network. Programs run under Windows XP with a CPU clock rate of 1.73 GHz and 1GB in main memory. Data servers are hosted on Windows XP operating system with an Intel Core2 Duo processor having 3 GB in main memory and a CPU clock rate of 2.26 GHz. We used the following RDBMS: MySQL 5.0, PostgreSQL 8.1 and Microsoft SQLServer 2005. The database is a synthetic instance of the TPC-H schema, generated with various size. In the following, *Dataset(n)* denotes an instance with *n* customers and 2**n* parts. Each instance follows a simple distribution which associates two orders to each customer and two lineitems to each order. *n* parts are involved in the association between Lineitem and Part.

We report the experiments for three programs over this schema, P_{simple}, P_{cust} and P_{part}, whose summaries are shown on Figure 10. In order to validate our choices, we compare the result of the following execution strategies: Lazy (an execution similar to a cursor-based implementation); Single (the data graph is materialized with a single query that (outer) joins all the relations accessed by the program), and Optim, our optimized strategy which combines outer joins and decorrelation.

5.2 Constant cost of open and fetch

Recall that our cost model assumes that the costs of **open** and **fetch** are constant, and thus independent from, respectively, the query size or the row size. In order to validate these assumptions, we first extracted the following SQL queries from the P_{cust} program, over the *Dataset(20K)* SQLServer database:

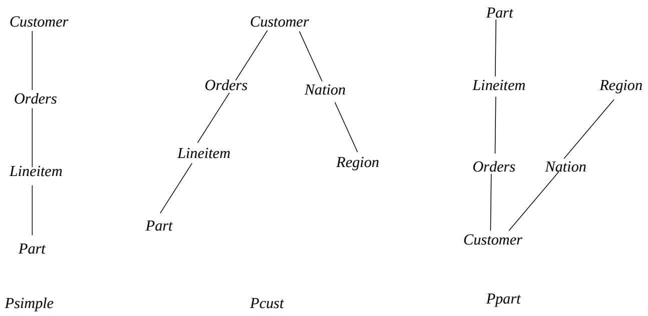

Figure 10: Summaries of the P_{simple}, P_{cust} and P_{part} programs

```
Customer query:
  select max(c.acctbal) from Customer c

Part decorrelated query:
 select max(p.container) from part p where p.partkey in (
  select l.partkey from lineitem l where l.orderkey in(
      select o.orderkey from orders o where o.custkey in (
        select c.custkey from customer c)))

Outer join query:
  select max(c.acctbal) from Customer c
     left outer join Order o on c.custkey = o.custkey
       left outer join lineitem l on o.orderkey = l.
           orderkey

Single query:
    select max(c.acctbal) from Customer c
     left outer join Order o on c.custkey = o.custkey
       left outer join lineitem l on o.orderkey = l.
           orderkey
             left outer join part p on p.partkey=l.partkey
```

We measure the execution of each query on the server, independently from communication and decoding overhead. Table 2 gives the results. The costs obviously differ with respect to both the complexity of a query expression and the specific system. With SQLServer, running a single query which retrieves *all* the data needed by a program is 25 times more costly than running a simple query than scans the Customer table (but retrieves only customer information). The ratios become respectively 50 and 77 for PostgreSQL and MySQL. Nevertheless, they appear negligible with respect to the number of queries executed by a program that relies on the Lazy strategy. An invoice program that scans the Customer table for instance must execute 20,000 queries to retrieve the Order rows, and much more to retrieve the LineItem rows. We conclude that the differences in query execution time on the server are negligible with respect to the N+1 query problem and communication overhead.

Next, we analyse the impact of the row size of the **fetch** cost. We repeatedly execute a program that fully scans a table with 1,000 rows. For each scan we

	Cust Q.	Part Q.	Outer join Q.	Single Q.
SQLServer	0.016	0.281	0.391	0.412
PostgreSQL	0.031	1.344	0.532	0.828
MySQL	0.016	0.281	1.14	1.245

Table 2: Cost of executing **open** (server side only)

vary the size of the row created by the **select** clause. Table 3 shows the results. It turns out that retrieving large rows does not make a great difference. This can be related to the default packet size (Maximum Transmission Unit) of the underlying network protocol. For ethernet (LAN), the max packet size is 1500 octets. The same argument as before holds: a program retrieving 1,000 rows of 1,000-bytes each will be about two times faster than a program that gets 2,000 rows of 500 bytes. In terms of database programming, this clearly calls for an execution of (outer) joins, putting rows from several tables together and minimizing the number of fetches.

Row size	50	100	300	500	800	1000	2000
Exec.time	2.8	2.9	2.9	2.95	2.968	3.078	3.359

Table 3: Impact of row size on **fetch** (client-server communication)

5.3 Relative costs of open, fetch and decode

We analyse independently the impact of the number of **open** or **fetch** on the program execution. We first execute a program with a unique query that scans a table with 10,000 rows. The number of **fetch** is thus constant (10,000). We vary the number of **open** from 1 (the query addresses all the rows) to 10,000 (the query is executed for each row). Results are shown in Table 4 and Figure 11. The figures for the first column represent the cost of fetching and decoding the data, with only one query execution. The remaining columns represent the overhead of executing many **open**. Next, we scan the same table, executing only one query, but stopping the loop after *n* **fetch**. Table 5 and Figure 11 show the results.

The linear aspects of the curves confirm our expectations. The slope in Figure 11 (constant C_{open} in Equation 1) is almost 1 for the three database servers. The obvious interpretation is that queries are executed independently on the server, therefore the cost of submitting *n* queries is *n* times the cost of submitting only one query. The constant associated to **fetch** is rather 0.3-0.5 and depends on the database system. This can be explained by low-level session cache effects, either on the server or on the client, along with packet clustering during network exchanges.

5.4 Comparison of materialization strategies

Table 6 shows the execution time of P_{simple}, P_{cust} and P_{part} programs running over the *Dataset(20K)* database. We report the figures for the Lazy, Optim and Single strategies. Executing independently a lot of small queries leads to dramatic execution times. The results are consistent for the three RDBMS used, and confirm the major impact on the performances of the client/server exchanges and network round-trips. Optim and Single consistently outperform Lazy by one or even two

Nber of opens	1	500	1,000	2,000	2,500	5,000	10,000
SQLServer	5.063	5.985	6.75	8.281	8.672	16.875	45.14
PostgreSQL	3.703	12.969	22.344	42.921	49.719	96.094	208.531
MySQL	3.515	5.125	5.609	7.329	8.515	16.203	36.734

Table 4: Impact of the number of **open** on the program execution

Nber of fetches	1,000	2,000	3,000	4,000	5,000	7,000	8,000	9,000	10,000
SQLServer	2.875	3.156	3.329	3.438	3.781	3.891	4.046	4.297	4.375
PostgreSQL	2.078	2.375	2.484	2.625	2.735	3	3.141	3.313	3.5
MySQL	2.14	2.328	2.578	2.782	2.875	3.078	3.218	3.234	3.593

Table 5: Impact of the number of **fetch** on the program execution

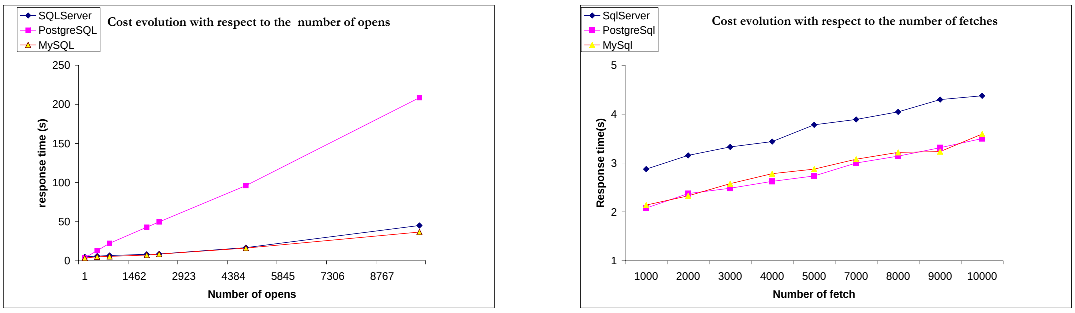

Figure 11: Cost evolution with respect to the number of opens and fetches

orders of magnitude, and have close execution times for the summaries P_{simple} and P_{cust}. Interestingly, the Optim strategy becomes faster than Single for P_{part}. The reasons lies in the number of redundant decoding carried out by P_{part}.

Consider the summary of P_{part} in Figure 10. Single executes a query with 5 outer joins associating the 6 tables. Since the dataset contains 80,000 line items, the program performs 80,000 fetches, each retrieving one large row with 6 tuples. Redundant decoding is performed for orders, customers, nations and regions. There are for instance 30 distinct nations represented in the result, but one nation appears in each of the 80,000 rows. A same nation is therefore transmitted 2,666 times on average!

Optim avoids this overhead. Region, Nation, Customer and Order are successively put in the cache. Then an outer join in the first block Part-LineItem is executed, leading to 80,000 fetches, but no redundant decoding.

Figure 12 shows the behavior of the execution strategies by varying the database size. Both Single and Optim avoid the N+1 query pitfall, and enjoy a linear increase of execution time. Lazy, on the other hand, executes a number of queries which is exponential in the depth of the query summary. We use a modest database with a few thousands rows in each table. The Lazy technique would clearly not scale to very large datasets.

In summary, our simple cost model leads to an effective and robust improve-

	P_{simple}			P_{cust}			P_{part}		
	Single	Optim	Lazy	Single	Optim	Lazy	Single	Optim	Lazy
$SQLServer$	36	34	1,527	37	35	1,947	44	37	2,103
$PostgreSQL$	58	48	7,345	67	49	9,321	82	57	11,698
$MySQL$	37	35	1,678	38	36	1,989	50	39	2,302

Table 6: Execution time of P_{simple}, P_{cust} and P_{part}, in seconds

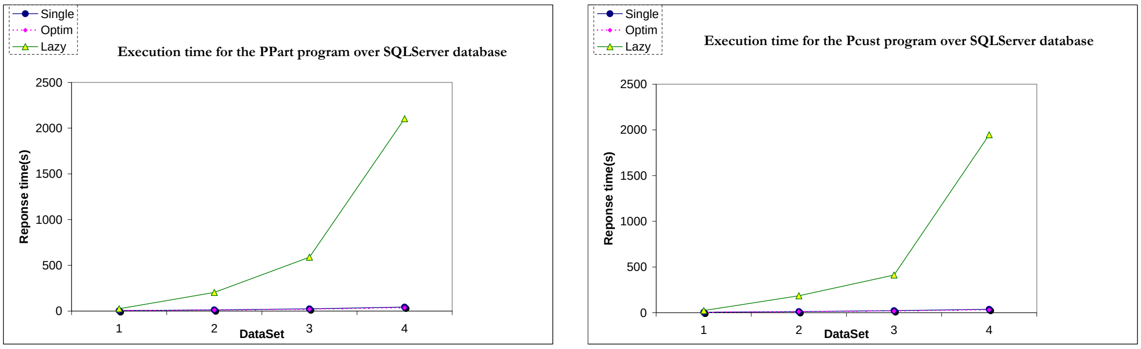

Figure 12: Response time of the P_{cust} and P_{part} programs execution over a SQLServer database

ment of database program execution. Executing a tree of single queries with a naive nesting of cursor-based table scan is a non-scalable solution. We considered the brute-force rewriting of the whole program summary as a single query, and showed that results in a linear behavior of the program execution, roughly proportional to the number of fetched rows. Our Optim algorithm is a simple technique to further improve this result, by getting rid of the redundant decoding.

6 Related work

Our work is related to studies aiming at publishing relational data as XML views [6, 17, 13, 9, 4]. These middleware systems specify a view-definition language in order to map between XML and relational data: application queries expressed in terms of semi-structured data are transformed into SQL queries. The SilkRoute system [9] relies on XQuery expressions over canonical XML virtual views [1]. SilkRoute uses a cost-based optimization algorithm, which obtains its cost estimates from the relational engine. The impact of client/server exchanges is not considered. The ROLEX system [4] provides a dynamic, virtual DOM interface to relational data, and supports DOM operators which are converted on the fly in SQL queries. ROLEX optimization cost model is based on a navigation profile (adopting, like us, a very simple model for this). ROLEX query evaluation strategies are customized for navigational access, i.e., for lazy materialization. Moreover, the context is different from ours, since a user can navigate freely in the (virtual)

XML document, whereas the behavior of databases programs is much more predictable. This is true as well for [13], which focuses on lazy execution, and [2] which proposes an incremental evaluation strategy.

Decorrelation strategy is often considered for query optimization. The aim of the decorrelation problem is to dissociate the execution of a subquery block from that of a correlated outer block. Our problem (materialization strategy) consists in choosing an order of data materialization in a graph, an edge $v_1 \rightarrow v_2$ in this graph beeing a kind of correlation between v_1 and the subgraph rooted in v_2. This problem has been widely studied in the context of SQL queries (see [12, 8, 14]). The specification of the exported data as a tree of co-related SQL queries can be viewed as an abstraction of nested cursors over result sets, each defined with respect to its parents [3, 7]. Query rewriting techniques are mostly viewed in terms of database server optimization which aims, basically, at limiting I/O operations [10, 16]. As shown in the present paper, we can transpose these techniques to a client/server context where the network cost is considered as predominant.

Finally, a close work is the Scalpel system [5]. Basically, Scalpel observes the flow of queries submitted by an application, attempts to build query patterns (typically, joins built from nested queries), and then rewrites these patterns as query execution plans (e.g., merge join or hash joins) executed on the client side. The system relies on a sophisticated cost-based optimizer which estimates the efficiency of execution strategies with respect to many parameters, including the open and fetch operations. Our approach is quite similar in its motivation and its initial observations. It differs by several aspects: (i) we chose an explicit declaration of the program summary, letting an expert user decide the part of the program which must be optimized, (ii) our data model is based on a graph representation in order to support efficiently the object-based orientation of ORM layers, and (iii) we adopt a simple cost model, focusing on the reduction of query submissions which constitute the major part of the total cost.

7 Conclusion and Future Work

Our approach extends ORM principles with global optimization strategy. The design of our solution does not burden the programmer's task with complex optimization issues. Instead, an optional summary must be provided during the initial step of the program execution. The summary can be defined by an independent expert user, even after the design and implementation of an application. Failing to provide this information to a program simply results in a lazy materialization mechanism, where objects are loaded whenever they are needed. However, the results obtained by our strategy outperforms this standard execution by several order of magnitudes, with impressive improvements. The ratio between the low effort of defining a program summary and the high gain in terms of performance makes our approach a practical candidate for achieving physical independence.

Several extensions can be envisaged. First, our optimization techniques do not take into account the size of the client cache. If the cache cannot hold the whole program trace, then replacement techniques must be adopted. In the worse case a degenerated execution can be chosen, where large blocks are evaluated using the Lazy strategy which limits the memory requirements. Second, our summary constitutes a rather simple representation of a program behavior. This is not a problem, as long as it covers the costly part of a program execution. Extending the model accuracy, taking account for instance conditional statements, could nevertheless improve the optimizer choices. These extensions are part of future work.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[2] P. Bohannon, P. Buneman, B. Choi, and W. Fan. Incremental Evaluation of Schema-Directed XML Publishing. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 503–514, 2004.

[3] P. Bohannon, H. Korth, and P. Narayan. The Table and the Tree: Online Access to Relational Data through Virtual XML Documents. In *Intl. Proc. on WebDB 2001 Workshop on Databases and the Web*, 2001.

[4] P. Bohannon, H. Korth, P. Narayan, S. Ganguly, and P. Shenoy. Optimizing View Queries in ROLEX to Support Navigable Tree Results. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2002.

[5] I. T. Bowman and K. Salem. Optimization of Query Streams Using Semantic Prefetching. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 179–190, 2004.

[6] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 646–648, 2000.

[7] B. Choi, W. Fan, X. Jia, and A. Kasprzyk. A Uniform System for Publishing and Maintaining XML Data. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1301–1304, 2004.

[8] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 197–208, 1987.

[9] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Trans. on Database Systems*, 27(4):438–493, 2002.

[10] R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 23–33, 1987.

[11] HIBERNATE 3.3.0. Red Hat Middleware, 2004. http://www.hibernate.org.

[12] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Trans. on Database Systems*, 7(3):443–469, 1982.

[13] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. Navigation-driven evaluation of virtual mediated views. In *Proc. Intl. Conf. on Extending Data Base Technology*, pages 150–165, 2000.

[14] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 39–48, 1992.

[15] Ruby on the Rails. http://rubyonrails.org.

[16] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex Query Decorrelation. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 450–458, 1996.

[17] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *The VLDB Journal*, 10(2-3):133–154, 2001.

[18] D. Shasha and P. Bonnet. *Database Tuning: principles, experiments, and troubleshooting techniques*. Morgan- Kaufmann, 2002.

[19] TPC Benchmark H (Decision Support), Standard Specification, Revision 2.8.0. Transaction Processing Performance Council, 2008. http://www.tpc.org/tpch.

[20] The XPath language recommandation (1.0). World Wide Web Consortium, 1999. http://www.w3.org/TR/xpath.

[21] The Zend Framework. Zend Technologies. http://framework.zend.com.